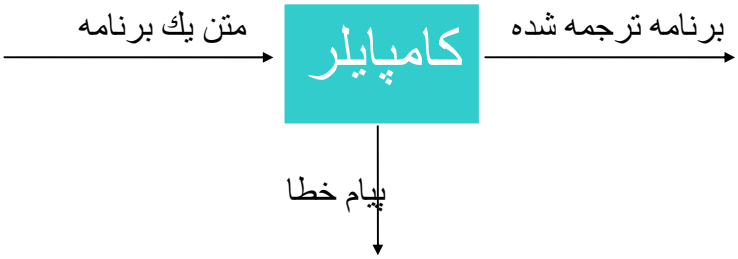
فصل اول

مقدمه

۱.۱ تعريف كامپايلر

كامپايلر برنامه اي است كه در ورودي خود، متن يك برنامه را كه طبق قوانين ودستور زبان تدوين شده براي آن كامپايلر مشخص شده است پذيرفته و در خروجي برنامه اي به زبان دوم ايجاد مي نمايد. اين زبان دوم مي تواند ماشين يا برنامه به زبان ديگر باشد.



شكل 1- ورودي و خروجي كامپايلر

برنامه ترجمه شده در بعضي از محيط ها مثل Unix معمولا" به زبان C ميباشد . براي مثال كامپايلر راتفورد كه نوع پيشرفته تر از فورترن ميباشد در خروجي خود برنامه به زبان C را ايجاد مي نمايد كه توسط كامپايلر C تبديل به كد ماشين مي گردد.

بعضي از كامپايلر ها كه در اصطلاح *مفسر* يا Interpreter ناميده مي شوند همگام با ترجمه هر جمله اجرايي ، آن را به اجرا در مي آورند. براي نمونه زبان نرم افزار Dbase را ميتوان نام برد. البته اين ترجمه همراه با اجرا، زمان اجرايي را بسيار طولاني ميكند، اما امكاناتي در اختيار برنامه نويس قرار ميدهد كه كامپايلرهاي عادي قادر به حصول آن نيستند . براي مثال در زبان Dbase دستورالعمل ماكرو كه با عملگر & مشخص مي شود اين امكان را فراهم ميكند كه بتوان در زمان اجرا محتوي يك رشته را به اجرادرآورد . به مثال زير توجه نمائيد:

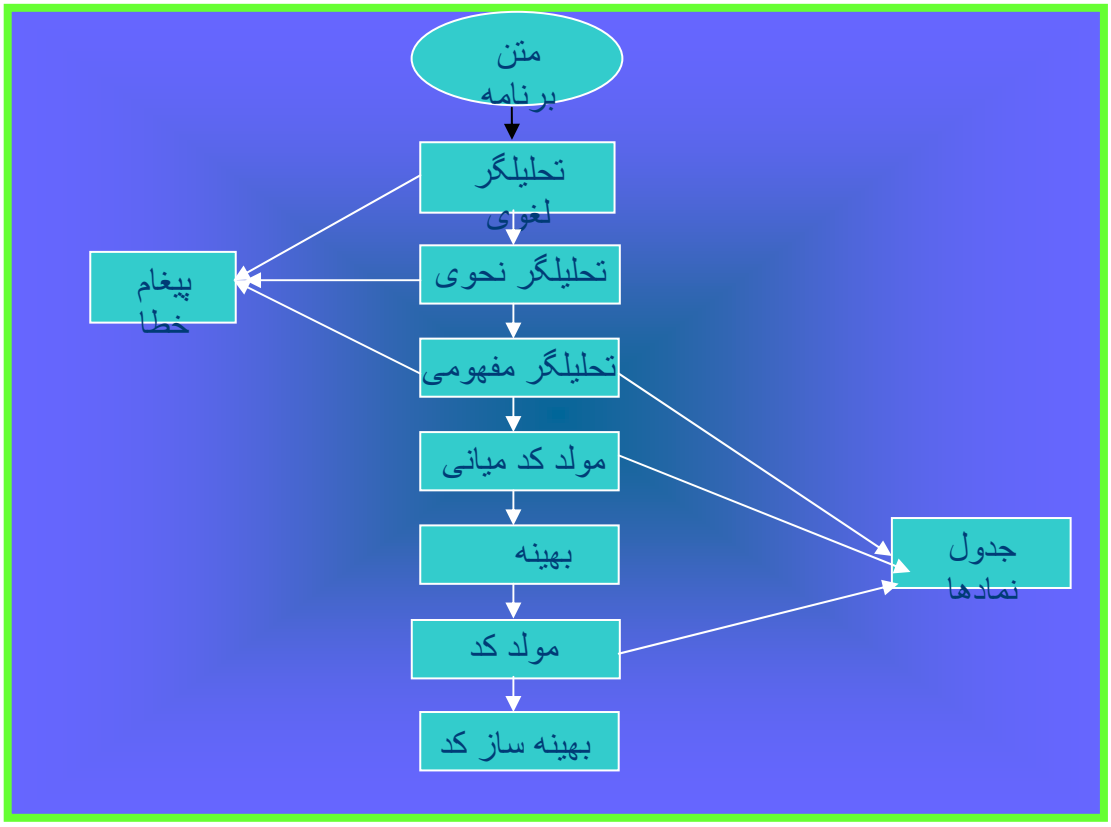
```
A = 'B = 10'
&A
```

```
@ 2,3 get A
&A
```

در مثال فوق با اجراي اولين دستورالعمل &A محتواي A بعنوان يك دستورالعمل تلقي مي شود وبه اجرا در مي آيد و به اين ترتيب محتواي B برابر 10 قرار ميگيرد . عمل كامپايل در طي مراحل مشخص كه در بخش بعدي در مورد آن صحبت خواهد شد انجام ميگردد. اين مراحل ميتوانند بطور همزمان اجرا شوند.

۱.۲ مراحل كامپايلر

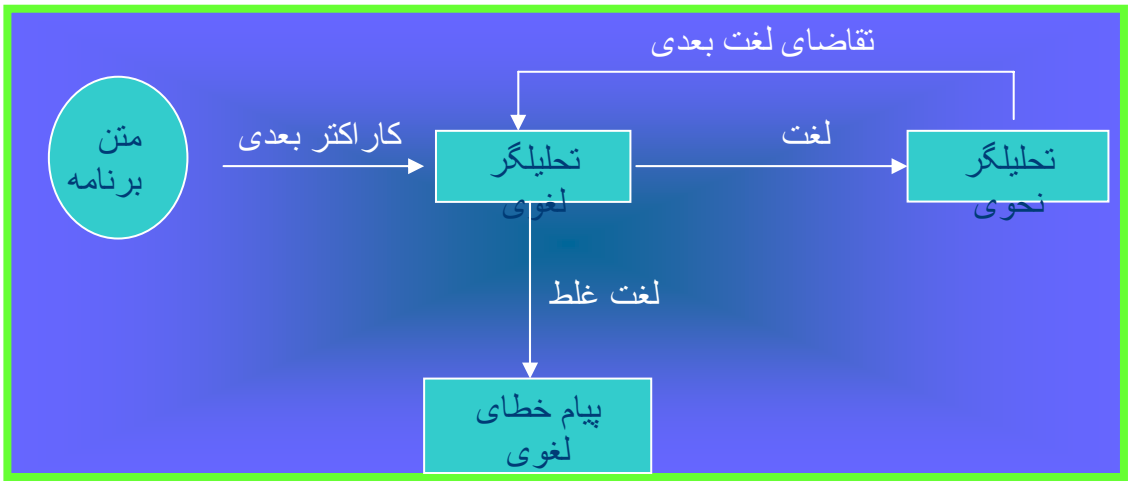
همانطور كه در شكل 1.2 مشخص شده است، كامپايلرها در حالت كلي از هفت بخش اصلي تشكيل مي شوند . در اين بخش بطور خلاصه بخشهاي متفاوت كامپايلر تشريح ميشوند.



شكل 1.2- ساختار كلي كامپايلرها

الف ـ تحليلگر لغوي

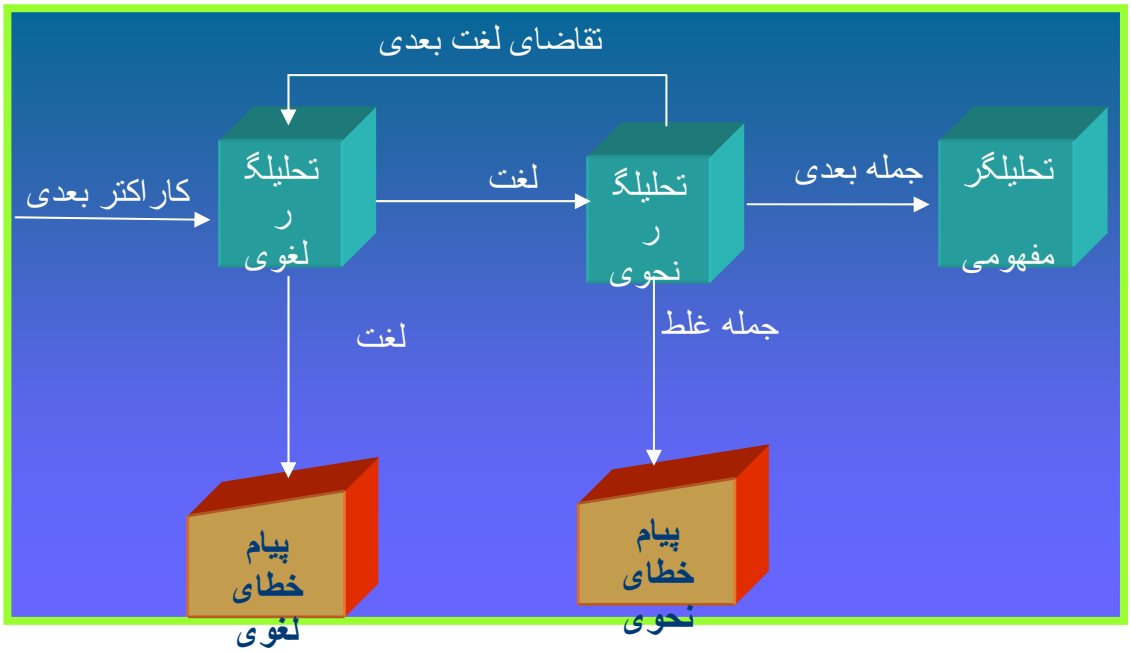
وظيفه تابع ***تحليگر لغوي*** يا در اصطلاح خارجي Lexical Analyser ، تشخيص لغات در متن برنامه مورد كامپايل است كه توسط بخش ديگري از كامپايلر تحت عنوان تحليلگر نحوي مورد فراخواني قرار ميگيرد. با هر بار فراخواني اين لغت بعدي را از متن برنامه تشخيص ميدهد و اطلاعات لازم در مورد لغت را براي تحليلگر نحوي ارسال ميدارد. هر زبان برنامه سازي قوانين لغوي مربوط بخود را دارد . قوانين لغوي بيانگر فرم كلي انواع لغات در زبان برنامه سازي است . تحليلگر لغوي در صورت وجود خطا در قالب بندي لغات استفاده شده در متن برنامه مورد كامپايل اعلام *خطا لغوي* مينمايد. تابع تحليلگر لغوي در فصل دوم از اين كتاب مورد بررسي واقع خواهد شد.



شكل 1.3- شماي تحليلگر لغوي

ب - تحليلگر نحوي

وظيفه ***تحليلگر نحوي*** يا در اصطلاح Syntax Analyser ، تشخيص صحت فرم ظاهري برنامه ها از لحاظ دستورالعمل زبان برنامه سازي مربوطه است . تحليلگر نحوي با فراخواني تحليلگر لغوي لغات را از متن برنامه مورد كامپايل دريافت و صحت قرار گرفتن آنها را در مجاور يكديگر بر اساس دستورالعمل زبان مربوطه مورد آزمون و تحليل نحوي قرار ميدهد. در صورتيكه از لحاظ دستورالعمل زبان ، برنامه مورد كامپايل داراي خطا باشد پيام خطا توسط تحليلگر نحوي صادر ميگردد .

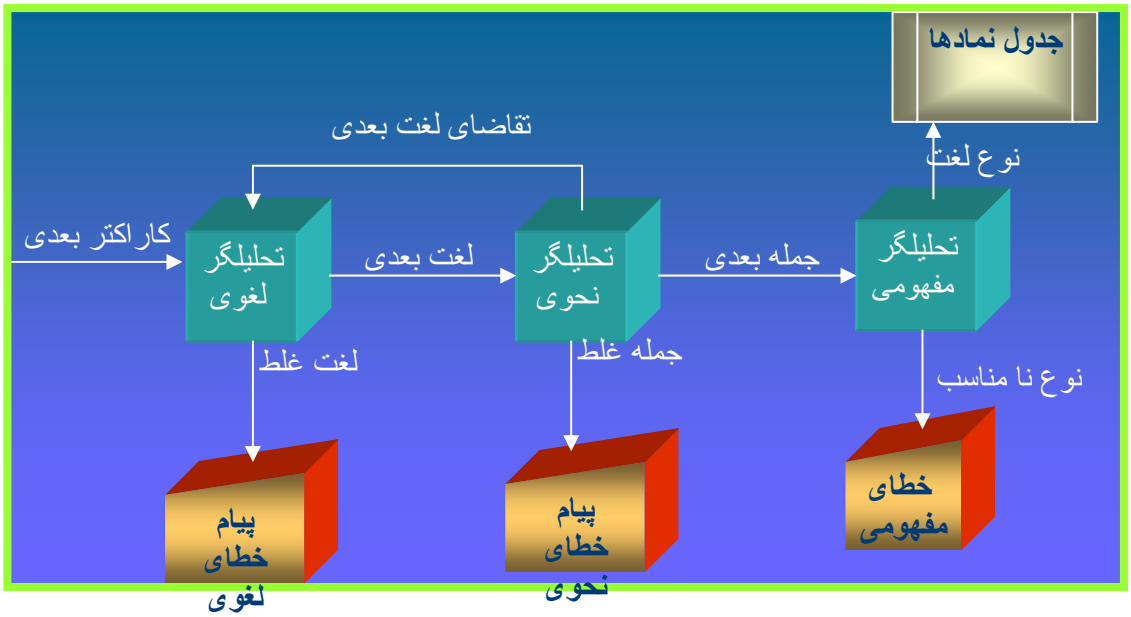


شكل 1.4- شماي كلي تحليلگر نحوي

انواع روشهاي تحليل نحوي و تشخيص صحت برنامه ها بر اساس دستورالعمل و گرامر زبانها برنامه سازي در فصل هاي 3 ، 4 و 5 مورد بحث و بررسي واقع خواهد شد .

ج- تحليلگر مفهومي

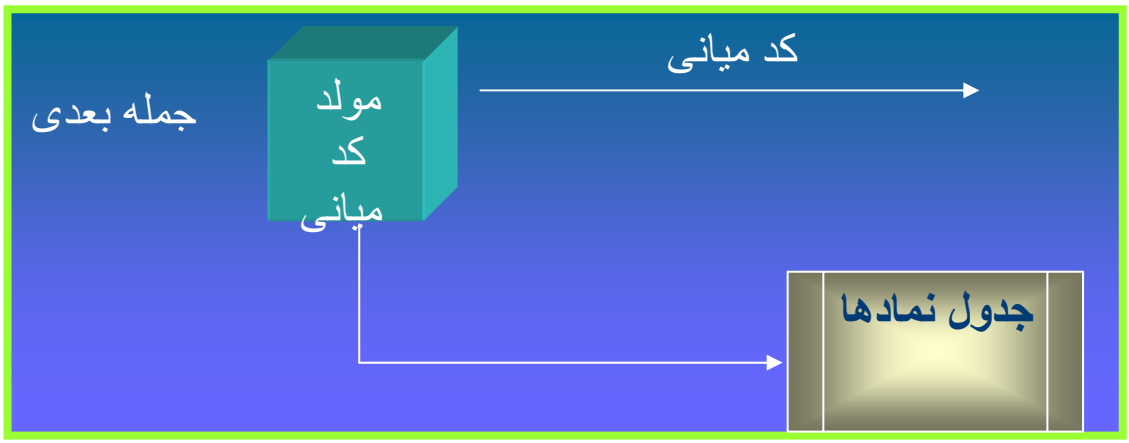
كار تحليلگر مفهومي يا در اصطلاح Semantics Analyser تعيين صحت مفهوم جملات است. ممكن است يك جمله ازنظر نحوي صحيح ولي از لحاظ مفهومي داراي خطا باشد. براي مثال جمله مهردادآتش خورد عليرغم اينكه در دستور زبان فارسي از نظر نحوي صحيح ميباشد از نظر مفهومي نادرست است . در كامپايلرها معمولا" عمل تحليل مفهومي محدود به آزمون نوع يا Type Checking است . تحليلگر مفهومي وابسته به نوع اسامي و متغيرها كه در جدول نمادها مشخص شده، صحت استفاده آنها را جملات و عبارات مختلف مورد آزمون قرار ميدهد. براي مثال اگر متغيري از نوع رشته تعريف شده باشد نمي توان آن را با متغيري از نوع صحيح يا اعشاري جمع و يا مقايسه نمود. فصل 7 در ارتباط با آزمون نوع است.



شكل 1.5- شماي تحليلگر مفهومي

د- مولد كد مياني

مولد كد مياني بخشي از كامپايلر است كه در ورودي خود جملات تشخيص داده شده توسط تحليلگر نحوي را پذيرفته و در خروجي كد واسطه يا مياني توليد ميكند. كد مياني بسادگي قابل تبديل و نزديك به زبان ماشين است اما مستقل از ساختار و جزئيات هر گونه ماشين خاص ميباشد .



شكل 1.6- شماي كلي مولد كد مياني

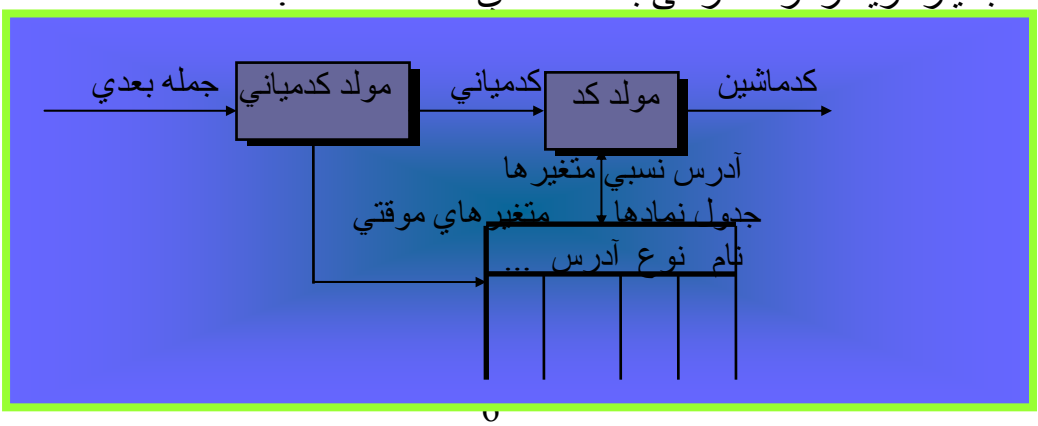
يكي از مزيتهاي ايجاد كد مياني افزايش قابليت حمل كامپايلر بر روي سخت افزار ها با كد متفاوت و ساختار متفاوت است . به اين ترتيب كه برنامه كامپايلر را بتوان بروي كامپيوترها با كد ماشين و اسمبلي متفاوت كامپايل نموده ، با اجراي آن متن برنامه هاي مورد كامپايل را به كد مياني تبديل كرد و سپس با نوشتن يك برنامه كوچك اين كد واسطه را به كد ماشين مورد نظر تبديل نمود. مزيت ديگر ايجاد كد مياني ، فراهم نمودن شرايط خوب براي بهينه سازي كد برنامه ها است. كد مياني در فصل 6 مطرح شده است.

ه- بهينه ساز كد مياني

هدف از بهينه سازي كد مياني ميتواند تقليل حجم و افزايش سرعت اجرايي كدماشين حاصل از كار كامپايلر باشد . براي اين منظور بهينه ساز ، كد مياني حاصل از مولد كد مياني را مورد تحليل قرار مي دهد. معمولا" تحليل كد مياني با آزمون نمادي برنامه ها تحقق مي يابد ، به اين ترتيب كه مفهوم برنامه ها در زمان كامپايل، با تبديل كد مياني به يك گراف بنام گراف جريان مورد بررسي و تحليل واقع شده ، سعي ميكنند تا حجم كد حاصل را تقليل داده ، در صورت وجود كد زائد را مشخص و حذف نمايد. بهينه سازي موضوع فصل 8 از اين كتاب است. تحليل نمادي براي آزمون صحت نرم افزارهاي كاربردي در مباحث مهندسي نرم افزار نيز مطرح ميباشد . مبحث آزمون نرم افزار تحت عنوان Software testing شناخته شده است . تكنيكهاي بهينه سازي در تشخيص توازي كد برنامه ها و اجراي موازي (Parallel) يا هم زمان كد برنامه ها بروي چند پردازنده نيز مورد استفاده قرار ميگيرد .

و- مولد كد

اين بخش از كامپايلر وابسته به ساختار سخت افزار است كه معمولا" فازي جداگانه در كامپايلر ميباشد و كد مياني بهينه شده را تبديل به كد ماشين (Object Code) مينمايد . در الگوريتمهاي توليد كد، معمولا" هدف اينست كه حداكثر بجاي ارجاع به حافظه از ثباتهاي ماشين استفاده شود، چرا كه دسترسي به يك ثبات بسيارسريعتر از دسترسي به خانه هاي حافظه است .



شكل 2.7- شماي مواد كد

همانگونه كه در شكل فوق مشاهده ميشود در زمان كامپايل، آدرس نسبي متغيير ها و اسامي بكار رفته در داخل برنامه ، درون جدول نمادها ثبت ميگردد . اين آدرس هاي نسبي هميشه نسبت به شروع برنامه و در داخل هر زير برنامه نسبت به آدرس شروع زيربرنامه سنجيده ميشود. آدرس شروع براي هر زيربرنامه و برنامه اصلي در زمان كا مپايل صفر درنظر گرفته ميشود . در هنگام پيوند و قرار دادن برنامه ها در حافظه اصلي كه به منظور اجراي برنامه صورت ميگيرد آدرسهاي شروع محاسبه ميگردند . روشهاي توليد كد در فصل 9 مطرح شده است.

ز - بهينه ساز كد

هدف از بهينه سازي كد ، تقليل حجم اشغالي و تسريع كد اجرايي برنامه است . در اين مرحله براي انجام عمل بهينه سازي و رسيدن به اهداف آن ساختار اسمبلي و كد ماشين و امكانات سخت افزاري آن در نظر گرفته ميشود و از دستورالعمل هايي كه سريعتر و با حجم كمتر به اجرا در مي آيند استفاده مي شود بهينه ساز كد در صورت امكان دستورالعمل هاي سريع و كم حجم را با دستورالعمل هاي بكار برده شده در خروجي مولد كد، كه همان كد اجراي برنامه است جايگزين ميكند .

البته مراحل فوق الذكر بعضا" ممكن است در ايجاد كامپايلر حذف شود . براي مثال قسمت بهينه سازي و يا تحليل مفهومي ممكن است در يك كامپايلر وجود نداشته باشد . و نيز ممكن است كليه مراحل در يك مرحله و به صورت همزمان اجرا شود . معمولا" عمليات كامپايل برنامه ها در دو مرحله ويا در اصطلاح در دو Pass انجام مي شود .

۱.۳ كامپايلر كامپايلرها

كامپايلر كامپايلر در واقع يك مولد كامپايلر ميباشد و يا برنامه اي است كه در ورودي خود قوانين لغوي و گرامر و كدي كه براي هر جمله ، عبارت و عنصري از زبان را كه بايد توسط مولد كد ايجاد شود را در ورودي پذيرفته و در خروجي خود يك كامپايلر ايجاد ميكند .

در طي فصل هاي بعدي خواهيم ديد كه چگونه ميتوان يك مولد تحليل گر لغوي و تحليل گر نحوي را بسادگي توليد نمود . از كامپايلر كامپايلرهاي رايج يكي YACC را ميتوان نام برد. اين مولد كامپايلر قواعد گرامري و كد مورد نظر

براي تبديل و در واقع كامپايل جملات متن برنامه كامپايلر را در ورودي خود دريافت ميكند. خروجي YACC برنامه كامپايلر به زبان C است . عمل تحليل لغوي براي اين كامپايلر كامپايلر توسط يك مولد تحليلگر لغوي بنام LEX انجام ميشود. در اين كتاب نشان داده خواهد شد كه چگونه ميتوان يك تحليل گر لغوي ايجاد نمود كه بسيار ساده تر از LEX بتوان آنرا براي تحليل لغوي مورد استفاده قرار داد.

۱.٤ كامپايلر برنامه ها با لغات فارسي

در زبان فارسي بعلت اينكه عبارات از چپ به راست و جملات از راست به چپ نوشته ميشوند ، نميتوان بسادگي با استفاده از روشهاي جاري كامپايلر زبان برنامه سازي كه لغات در آن به زبان فارسي هستند، ايجاد نمود. اگر دستورالعمل هاي زبان برنامه سازي با استفاده از لغات فارسي از سمت چپ به راست نوشته شوند اين مشكل برطرف خواهد شد.

ميتوان برنامه هاي C يا پاسكال و هر زبان برنامه سازي ديگري را با استفاده از لغات فارسي و به زبان فارسي نوشت و با استفاده از يك تحليلگر لغوي و جدول لغات كليدي فارسي و مشابه آن در انگليسي به زبان اصلي تبديل نمود. پس از اشكالزدايي و تكميل برنامه ها ميتوان بالعكس عمل نموده متن انگليسي را به فارسي تبديل نمود.

۱.٥ دلايل تأليف اين كتاب

زبانهاي برنامه سازي ابزار اصلي براي بهره بري از تسهيلات كامپيوتر و كامپايلرها ابزار توليد زبانهاي برنامه سازي هستند. لذا، كامپايلرها يكي از مباحث اصلي و علم مادر در زمينه نرم افزار است. نرم افزار و تكنولوژي كامپيوتر امروزه كليه علوم را تحت الشعاع خود قرار داده است. لذا، ايجاد و صدور قطعات نرم افزاري مي تواند بعنوان يك منبع درآمد و سرمايه ملي مطرح باشد.
پيشرفت در اين مسير بسادگي امكان پذير است زيرا، توليد قطعات نرم افزاري در سطح بازار جهاني نيازي به پشتوانه قوي تكنولوژيك و صنعتي ندارد.
متأسفانه رقابت جهاني موجب حذف مطالب اصلي و كاربردي از داخل كتب علمي بخصوص در زمينه نرم افزارشده است. تا حدي كه مشاهده ميشود اكثر كتب درسي در عمل كاربردي ندارند. كامپايلرها نيز از اين قاعده مستثني نيست. براي نمونه مرجع 795 صفحه اي اصلي كامپايلرها [5] كه در سال 1986انتشار يافته در

عمل مطالبي از قبيل بهبود از خطا[1] و طريق عملي پياده سازي يك كامپايلر را كتمان ميكند. در اين كتاب نيز هيچگونه مطلبي در مورد يكي از روشهاي رايج براي تجزيه گرامري جملات در برنامه هاي مورد كامپايل ، بنام كاهينه بازگشتي يا در اصطلاح خارجي Recursive Descent به ميان نيامده است.

مسئله بهبود از خطا در مراجع [6] ، [8] و حاصل تحقيقي كه بر روي چند مقاله در اين زمينه انجام شد بطور عملي مشخص شده و در فصول 4 و 5 از اين كتاب گنجانده شده است. روش كاهينه بازگشتي ومسإله بهبود از خطا در مرجع [9] مطرح شده است. در اين مرجع هچگونه مطلبي در مورد مسئله بهبود از خطا و توليد كد مياني ارائه نشده است.

روش كاهينه بازگشتي براي پياده سازي كامپايلر ساده اي بنام دلتا در مرجع [10] بصورت جامع و كامل تري مطرح شده است. كمبود اين كامپايلر، مسئله بهينه سازي يا در اصطلاح Optimiztion در كد ميباشد. در اين كتاب گرامر زبان دلتا تكميل تر شده ، مسإله بهينه سازي كد نيز در اين كامپايلر در نظر گرفته شده است. كد كامپايلر دلتا بطور كامل در انتها ارائه شده است.

اصولا"، بهينه سازي [13, 15, 17, 18] در كامپايلرها در جهت افزايش سرعت اجرايي برنامه ها و اندازه اشغالي از حافظه است. در مرجع [13] روشي جديد و عملي براي پياده سازي الگوريتمهاي بهينه سازي در كد برنامه ها ارائه شده است. در فصل 8 از اين كتاب روشي جديدتر و سريع براي پياده سازي الگوريتم هاي بهينه سازي مطرح شده است. اين روش بطور عملي در كامپايلر دلتا پياده سازي شده است.

تعيين فرم كلي لغات در زبانهاي برنامه سازي بطور خلاصه و بدون ابهام توسط نوعي خاص از گراف بنام ماشين خودكار يا Finite Automata و نوعي خاص از عبارات بنام عبارات با قاعده بيان ميشود. ايجاد ماشينهاي خودكار براي بيان ساختارن لغوي زبانهاي برنامه سازي در مرجع [16] به طرز مطلوبي بيان شده است. اين روش در فصل دوم از اين كتاب تحت عنوان تحليل گر لغوي يا Lexical Analyxer مطرح شده است.

[1] با مشاهده يك جمله كه داراي اشتباه از لحاظ دستورالعمل زبان برنامه نويسي است ، كامپايلر نيز در تشخيص جملات بعدي ممكن است موجب اشتباه شود. لذا، بهبود از خطاء يا Error Recovery براي پياده سازي كامپايلرها در عمل بسيار مهم ميباشد.

مراجع

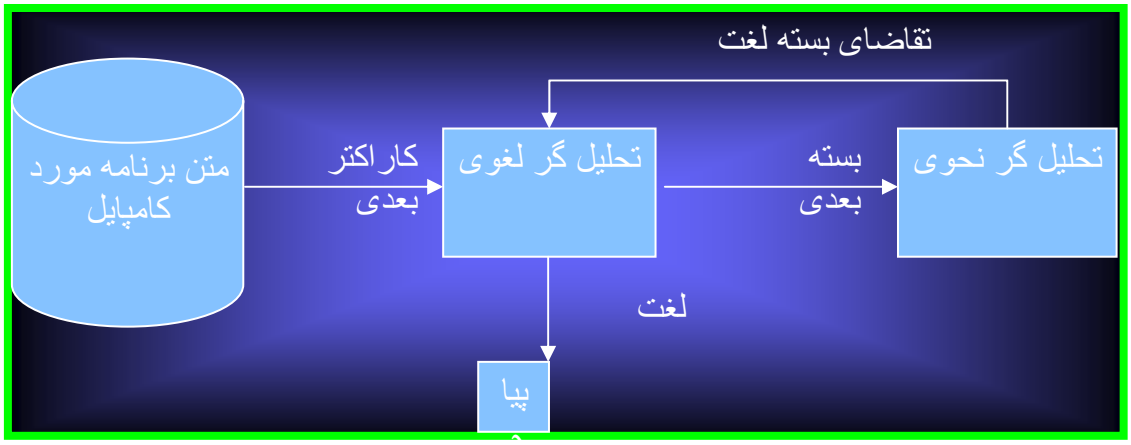
1- “SuperCompilers for Parallel and vector Computers”, Hans Zima, Acm Press, 1990, 376 pages.
2- “Parallel Programming and Compilers”, Constatine D. Polychronopoulos, Kluwer Academic Publishers, 1988, 230 Pages
3- “Symbolic Analysis For Parallelizing Compilers”, Mohamad R. Hagigat, KAP Publisher, 1885, 196 Pages.
4- “Interprocedural Def-Use Associations For C systems With Single Level Pointers”, Hermit D. Pande, IEEE Transaction On Soft. Eng., vol20, No.5, May 1994.
5- The Theory and Practice of Compiler Writing”, Paul G. Sorenson, Mc GrawHill, 1985
6- “Compilers Principles , Techniques and Tools”, Alfred V. Aho”, Addison- Wesley, 1986.
7- “Design of Compiler Techniques of Programming Language Translation”, Karen A. Lemone, CRC Press, 1992.
8- “Generation of Interactive Parsers With Error Handling”, E. Steegmans, IEEE Transaction on Software Engineering, vol. 18, no. 5, 1992.
9- “Brinch Hanson on Pascal Compilers”, Brinch Hanson, Printice-Hall, 1985.
10- “Programming Language Processors”, C.A.R Hoare, Printice-Hall, 1993.
11- “Compiler Design in C”, Allen L. Hollub, 1990.
12-“Object Oriented Compiler Construction”, Jim Holems, Printice Hall, 1995.
13- “ The Art of Compiler Design”, Thomas Pittman, Prentice-Hall, 1992.
14-“Introduction to Compilers”, Thomas W,parsons , Computer Science Press, 1992.
15-“Optimizing Schemes For Structured Programming Language Processors” , Tatsou Tsuji, Ellis Horwood, 1994.
16- “Compiler Construction, Theory and Practice”, W. A. Barrel, Science Research Associates INC, 1986.
17- “Interprocedural Def-Use Associations for C Systems with Single level Priorities. H. D. Pande, IEEE Transaction \s on Soft. Eng. , vol 20, may 1994.
18-“ Analysing the optimiztion techniques Compilers use to Transform your C Code”, Microsoft Systems Journal, March 1991.

فصل دوم

تحليلگر لغوی

۲.۱ مقدمه

تحليلگرلغوي تابعي است كه در ورودي خود برنامه مورد كامپايل را بعنوان يك فايل متن بازگشوده و كاراكتر به كاراكتر مي خواند وبا رسيدن به يك كاراكتر جداكننده لغت را تشخيص و تفكيك مينمايد. جداكننده ها مثل : Blank , Tab يا New line فقط به منظور جداسازي لغات بكار ميروند. دسته ديگر از جدا كننده ها مثل Semicolon يا + ارزش لغوي دارند. تحليلگر لغوي در خروجي خود اطلاعات مربوط به لغت را در يك ركورد يا ساختار بنام بسته لغت قرارمي دهد بسته لغت شامل اطلاعاتي در مورد لغت تشخيص داده شده توسط تحليلگر لغوي مانند سطر ، ستون و نوع لغت است بسته لغت را در اصطلاح Token گويند.



شكل 2.1- شماي كلي تحليلگر لغوي

هر زبان برنامه سازي قوانين لغوي خاص خود را دارد. قوانين لغوي زبانها را ميتوان بصورت غير مبهم و خلاصه در قالب نوعي خاص از عبارات بنام عبارت با قاعده بيان نمود. قوانين لغوي را نيز ميتوان توسط نوعي خاص از ساختارگراف بنام ماشين هاي خودكار مطرح كرد. ماشينهاي خودكار ابزاري براي تبديل قوانين لغوي به كد برنامه هستند. چنانچه لغتي بنا بر قوانين لغوي زبان برنامه سازي ايجاد نشده باشد ، تحليلگر لغوي اعلام خطا مينمايد.

۲.۲ ساختار ورودی / خروجی

در داخل تابع تحليلگر لغوي برنامه مورد كامپايل به عنوان يك فايل متن دسترسي ميشود. اين فايل در ابتداي برنامه كامپايلر وخارج از محيط تحليلگر لغوي تعريف وگشوده مي شودو به عنوان يك پارامتر به تابع تحليلگر لغوي ارسال مي گردد. در قطعه كد ارائه شده در شكل 2.2 چگونگی فراخواني تابع تحليلگر لغوي نشان داده شده است..

```
                    Void  main ( int arg c , char  *argv [  ] )
{
    File   *InText ;
      if  (  argc  <  2  )
            {
              clrscr  (  ) ;
              textcolor  ( RED  ) ;
              gotoxy  ( 10 , 5 )
                 cprintf  ( “  نام برنامه مورد كامپايل فراموش شده است “ ) ;
               if  (  !getch ( )  )   getch  ( ) ;
                                        }
       InText  =  fopen ( argv [ 1 ] , “  R  “ ) ;
        while  ! feof ( InText ) ;
                {
                  Token-type   Token ;    .
                 Token =  Lexer ( InText ) ;
                                  .
```

شكل 2.2- فراخواني تابع تحليلگر لغوي

نقطه گنگ در مثال فوق TokenType و در واقع نوع خروجي تحليلگر لغوي است. معمولا" اطلاعات زير در بسته Token قرار داده مي شود :

```
Typedef  struct  Token
        {  int    ROW ;  //                    شماره سطر
           int    COL; //                     شماره ستون
             int  BLKNO ;  //                  شماره آشيانه اي
         int  BLKORD ; //  شماره  ترتيب بلاك در برگيرنده
            enum  Symbols  type ;  //          نوع لغت
             char Name [ 30 ] ;  //            خود لغت
        }  Token-type ;
```

شكل 2.7- ساختار بسته لغت

در ساختار ارائه شده براي تعريف لغت كه در اينجا بسته لغت يا Token ناميده شده است شماره سطر وستون ظاهر شده است. با استفاده از شماره سطر و ستون لغت ، در هنگام صدور پيام خطا كامپايلر قادر به بيان دقيق مكان خطاء خواهد بود. BLKNO شماره آشيانه اي بلاك در بر گيرنده يك لغت را مشخص مي كند. تنها نام لغات براي تعيين و تشخيص آنها از يكديگر كافي نيست بلكه ، جهت تعيين يك لغت نياز به شماره آشيانه اي بلاك در برگيرنده آن نيز هست . براي نمونه به قطعه كد زير توجه كنيد.

```
{
   int  I ;
   I=5 ;
        { int I ;
          I=6 ;
          printf ( " 2nd blk   %d " / I ) ;
        }
              printf ( " \n  1st blk  %d " / I ) ;
          }
```

در قطعه كد فوق همانگونه كه مشاهده مي كنيد اگرچه كه براي دو متغير يك نام I تعيين شده اما شماره آشيانه اي بلاك در بر گيرنده ، وجه تمايز اين دو مي باشد. گاهي اوقات شماره آشيانه اي بلاك نيز كافي نيست براي نمونه درقطعه كد زيراگرچه كه شماره آشيانه اي بلاكها يكي است اما دو متغيير وجود دارد.

```
{  int  I  ;
    I = I + 1
}
{  int  I  ;
    I = I + 3
}
```

شماره آشيانه اي با ورود به بلاك افزايش وبا خروج از بلاك كاهش مي يابد. با ظهور هر بلاك جديد شماره بلاك يا BLKORD يك واحد اضافه مي شود در مثال فوق وجه تمايز دو متغيير شماره ترتيب بلاك در بر گيرنده آن است .

به هر لغت يك نوع تخصيص داده ميشود. نوعي شمارش پذير يا در اصطلاح Enumenated . براي نمونه تحليلگر لغوي اگردر متن فايل ورودي عدد 123 را تشكيل دهد، در داخل فيلد Name عدد 123 و در داخل فيلد Type ، نوع آن كه يك عدد صحيح مي باشد را مشخص مي كند . اگر در متن ورودي ، جمله زير قرار گرفته باشد :

```
ABC := 123 ;
```

تحليلگر لغوي با اجراي دستورالعملي مثل :

```
NextChar = getc ( in-text ) ;
```

ابتدا كاراكتر **A** وسپس **B** وبعد از آن **C** را از متن فايل ورودي خوانده ، حالا با مشاهده Blank و در صورت عدم وجود Blank با مشاهده علامت كولون (:) ، خاتمه اولين لغت را تشخيص داده ، در داخل فيلد Name رشته ABC و در داخل فيلد Type ، نوع آن كه يك شناسه مي باشد را قرار مي دهد.

در فراخواني بعدي تحليلگر لغوي با Blank مواجه مي شود. ازآن چشم پوشي مي كند وبه جلو مي رود و علامت =: را به عنوان لغت بعدي تشخيص داده در فيلد Name ، رشته =: و در فيلد Type ، عملگر تخصيص را قرار مي دهد. البته شماره سطر وستون وبلاك در بر گيرنده هر لغت نيز مشخص مي شود.

نوع لغات در فيلد Type از ساختار TokenType مشخص شده است. اين فيلد از نوع شمارش پذير بنام Symbols تعريف شده است. در داخل نوع شمارش پذير يا در اصطلاح Enumerated بنام Symbols انواع ممكن لغات در داخل زبان مورد نظر مشخص ميشود. بايد توجه داشته باشيد كه با تعريف متغير از نوع شمارش پذير مجموعه اي از مقادير ثابت يا Constant تعريف ميشود. نوع Symbols در زبان C بصورت زير براي نمونه تعريف ميشود.

```
enum  Symbols { S_Program, S_Const, S_Eq, S_Semi, S_Id, S_No, S_Type,
                S_Record, S_End, S_Int, S_Real, S_Char, S_Array, S_String,
                S_Begin, S_Var, S_Colon, S_ParBaz, S_ParBast, S_Set, S_of,
                S_BrackBaz, S_BrackBast, S_Case, S_Pointer, S_ConstString,
                S_Function, S_Procedure, S_Begin, S_Dot, S_Comma,
                S_If, S_Then, S_Else, S_While, S_Do, S_Repeat, S_Until,
                S_For, S_Add, S_Sub, S_Div, S_Mul,  S_Mod, S_Lt, S_Le,
                S_Gt, S_Ge, S_Gt, S_Ne, S_Not, S_And, S_Or};
```

همانگونه كه مشاهده ميشود انواع مختلف لغات بايد در نوع Symbols گنجانده شود.

۲.۳ عبارات با قاعده

عبارت با قاعده[1] ، فرمي است چكيده وخلاصه براي بيان قوانين لغوي زبانها. شكل لغات درزبانهاي برنامه سازي داراي فرم كلي خاص است. بر اساس اين فرم كلي است كه انواع لغات از قبيل شناسه ها (اسامي) ، اعداد ، رشته هاي ثابت كاراكتري ، جملات تفسيري (Comment) وساير لغات از يكديگر تفكيك و تميز داده مي شوند.

۲.۳.۱ نمونه هايي از عبارات با قاعده

به عنوان نمونه تعريف شناسه ها در زبان **C** را در نظربگيريد. يك شناسه يا Identifire در زبان **C** حتما" بايد با يك كاراكتر آغاز گردد و در ادامه ممكن است به هر تعداد وبا هر تركيبي از ارقام صفر تا نه (0-9) ، حردف الفباي انگليسی (A-Z) و علامت خط زير يا در اصطلاح (UnderLine) ادامه يابد. به عنوان مثال اسامي :

A , B_ _ 1 , BAC2345_ _

ازلحاظ زبان **C** به عنوان اسامي (شناسه) ، شناخته مي شوند. ميتوان با استفاده از يك عبارت با قاعده به صورت زير، فرم كلي شناسه ها را در زبان **C** تعريف نمود :

Identifire : Letter (Letterr | Digit | '_') *
Letter : A | B | ..... | Z | a | b | ..... | z
Digit : 0 | 1 | 2 | ..... | 9

در عبارت ارااه شده براي شناسه ها ، علامت '*' نمايانگر تكرار صفر يا بيشتر مي باشد و علامت ' | ' علامت يا مي باشد. مي توان بصورت چكيده نيز Digit را تعريف نمود :

Digit : [0 ... 9]

به همين ترتيب ، حروف الفباي انگليسی را بصورت زير مي توان تعريف نمود :

Letter : [a ... z , A ... Z]

شكل كلي اعداد صحيح را ميتوان با استفاده از يك عبارت به اين صورت مشخص نمود:

Number : digit digit *
Digit : [0 ... 9]

بر طبق اين عبارت ، يك عدد صحيح با يك رقم بين صفر تا نه آغاز مي شود و به هر تعدادي رقم مي تواند در ادامه آن ظاهر شود. براي مثال :

0 , 5 , 0123

[1] Regular Expression

همانگونه كه مشخص است ، يك عدد حداقل داراي يك رقم بايد باشد يا به عبارت ديگر يك عدد از يك يا بيشتر ارقام تشكيل شده ، مي توان عدد را بصورت زير نيز تعريف كرد :

Number : digit +

در اينجا علامت '+' نمايانگرتكرار يك يا بيشتر مي باشد، يادآوري مي كنيم كه علامت ستاره '*' نمايانگر تكرار صفر يا بيشتراست.

اعداد مي توانند داراي علامت و يا بدون علامت باشند

Number : (+ | - | ε) digit +

در اين عبارت با قاعده ، علامت اپسيلون ε نمايانگر *عدم وجود* و يا *تهي* مي باشد . عبارت فوق به اين صورت خوانده مي شود :

يك عدد داراي علامت بعلاوه يا منها و يا ممكن است اصلا" علامتي نداشته باشد و بدنبال آن به تعداد يك يا بيشتر ارقام بين صفر تا نه ظاهر مي گردد.

اعداد را ميتوان به فرمي ساده تر با استفاده از اين قاعده كه هر عبارت اختياري مثل **r** بصورت **r ?** نمايش داده ميشود ، بصورت زير مشخص نمود :

Number : (+ | -) ? digit +

بعنوان نمونه اي ديگر از عبارات با قاعده ، فرم كلي رشته ها در زبان پاسكال را در نظر بگيريد. بايد توجه داشته باشيد كه در اين زبان چنانچه در وسطح رشته علامت كوتيشن ' نياز باشد ، مي بايست آنرا دوبار تكرار نمود. براي نمونه چنانچه جمله This is your's در خروجي مورد نظر باشد، ميتوان دستور Writeln (' This is your '' s ') ; استفاده نمود. لذا، فرم كلي رشته ها با عبارت با يك عبارت با قاعده بصورت زير بيان ميشود :

String : ' (Characters | ' ') * '

در عبارت فوق ، مجموعه Characters نمايانگر هرگونه كاراكتر قابل مشاهده منهاي كوتيشن است.

۲.۳.۲ قوانين ايجاد عبارات با قاعده

براي ايجاد يك عبارت با قاعده ، تعدادي از علائم مورد استفاده واقع مي شوند. هريك از اين علائم داراي معني و مفهوم خاصي مي باشد. در حالت كلي اگر دلتا (Δ) مجموعه علائم بكار رفته شده در عبارات باشد، آنگاه :

1. هر عنصر $a \in \Delta$ خود يك عبارت با قاعده است. براي مثال چنانچه $\Delta = [\ 0\ ..\ 9\ ]$ باشد آنگاه هر رقمي بين صفر تا نه مثل 1، خود به تنهايي يك عبارت با قاعده است.
2. چنانچه r و s دو عبارت با قاعده باشند ، آنگاه rs نيز يك عبارت با قاعده است ε به عبارت ساده تر در حالت كلي اگر دو عبارت با قاعده را در كنار يكديگر قرار دهيد ، حاصل يك عبارت با قاعده خواهد بود.
3. چنانچه r و s دو عبارت با قاعده باشند، آنگاه r يا s كه بصورت (r|s) مشخص ميشود نيز يك عبارت با قاعده است ε عملگر | داراي خاصيت جابجايي است ، به عبارت ديگر:
$$r|s = s|r = (r|s)$$
4. چنانچه r يك عبارت با قاعده باشد آنگاه r* نمايانگر تكرار صفر يا بيشتر از عبارات r است ε براي نمونه اگر r , a يا b باشد آنگاه r* برابر است با :
$$r : a|b$$
$$: (a|b)*$$
اين عبارت گوياى هر تركيبي با هر تعدادي از a يا b كه در كنار يكديگر قرار گرفته اند مي باشد. براي نمونه رشته هاي
AAA , BAABB , BBBB
همگى فرم هاي خاصي از عبارت r* هستندε بنابراين علامت اپسيلون ε كه نمايانگر عنصر تهي است ، به تنهايي يک عبارت با قاعده است
5. چنانچه r يك عبارت با قاعده باشد، آنگاه r^+ نمايانگر تكرار يك يا بيشتر عبارت r است. براي نمونه :
Number : digit $^+$
به اين ترتيب واضح است كه : $r^+ = (r^*|\varepsilon)$

با استفاده از پنج قاعده فوق ، مي توان عبارات با قاعده را بنا نمود.
با استفاده از نكات فوق ، فرم كلي جملات تفسيري (Comment) در زبان پاسكال را ميتوان بصورت زير معين نمود. بايد توجه داشته باشيد كه در زبان پاسكال جملات تفسيري در بين علائم آكولاد باز و آكولاد بسته قرار مي گيرند.

Comment1 : { C* }
C : (كليه كاراكترها منهاي آكولاد)

در زبان پاسكال جملات تفسيري را ميتوان بين علائم (* و *) نيز مشخص نمود. در اين فرم از جملات تفسيري بايد توجه داشته باشيد كه اگر ستاره در داخل جمله

ظاهر شود، در پشت آن بايد هر حرفي به غير از ستاره و پرانتز بسته ظاهر شودء براي نمونه ، لغت زير يك جمله تفسيري **نيست** :

(* abc *) acb *)

فرم كلي اينگونه جملات را ميتوان در قالب يك عبارت با قاعده بصورت زير خلاصه نمود:

Comment2 : '(' * ((r | * $^{+}$ s) * r) * * $^{+}$ ')'

r : كليه‌كاراكترها منهاي علامت *

s : كليه‌كاراكترها منهاي علائم * و ')'

به اين ترتيب جملات تفسيري بصورت زير

Comment : Comment1 | Comment2

تعريف مي شوند.

با توجه به اينكه اعداد يا به صورت صحيح و يا به صورت اعشاري ظاهر مي شوند و با در نظر گرفتن اينكه اگر قبل از ممیز حداقل يك رقم ظاهر شود آنگاه وجود ارقام پس از ممیز ضروري نيست و نيز اينكه اگر بعد از ممیز حداقل يك رقم ظاهر شود وجود رقم قبل از ممیز ضروري نيست . بطور خلاصه عبارت با قاعده براي بيان شكل كلي اعداد بصورت زير ميتواند باشد:

Number : (digit*.digit^{+}) | (digit^{+}.digit*) | digit^{+}

عبارات با قاعده مبين قوانين لغوي و نمايانگر فرم كلي لغات هستند. متاسفانه ، اين فرم كلي را بسادگی نميتوان تبديل به كد برنامه نمود. لذا ، براي رفع اين مشكل از ماشينهاي خودكار استفاده ميشود. .

۲.۵- ماشينهای خودكار

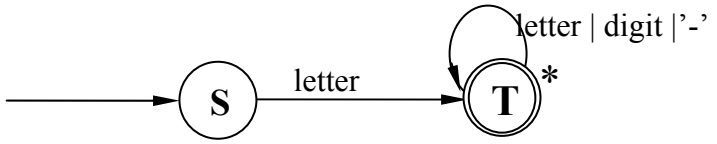
يك ***ماشين خودكار***[2] نوعي گراف مي باشدكه داراي تعدادي رئوس يا حالات (States) و تعدادي لبه يا يال (Edges) مي باشد و در اين گراف يالها خطوط واصل بين حالات هستند. اين ماشينها، ابزاري براي تعيين قوانين لغوي و تشخيص لغات مي باشند كه بسادگی قابل تبديل به كد برنامه بوده بطوري كه با استفاده از آنها مي توان تحليلگر لغوي را بصورت يك برنامه تبديل كرد و يا Source يك برنامه را توليد نمود.

براي نمونه شناسه ها را در نظر بگيريد. شناسه ها توسط عبارت با قاعده :

[2] Finite State Automata

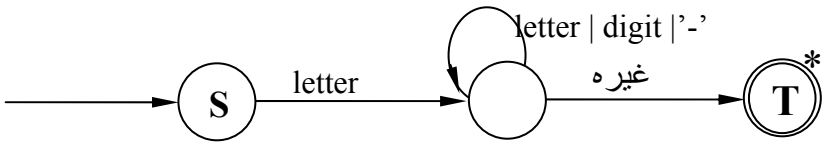
letter (letter | digit)*

در بخش قبل معين شدند. براي تشخيص شناسه ها مي توان ماشين خودكار زير را مورد استفاده قرار داد:



اينن ماشين خودكار را بصورت زير نيز ميتوان نمايش داد. بنابر دياگرام فوق در صورتيكه كاراكتر خوانده شده ازمتن فايل مورد تحليل لغوي يكي از حروف الفباي لاتين يا در اصطلاح letter باشد ميتوان وارد ماشين خودكار تشخيص شناسه ها شد. حالت شروع و در واقع نقطه ورود به ماشين خودكار با حرف S مشخص شده است. حالت بعدي T ناميده شده است. در حالت T اطمينان از تشخيص يك شناسه است زيرا ، يك حرف نيز به تنهايي يك شناسه در نظر گرفته ميشود. پس در اين حالت لغت خوانده شده بعنوان يك شناسه پذيرفته شده است.

حالت پذيرش ماشين خودكار را با دو دايره تو در تو نشان ميدهند. در حالت پذيرش T آنقدر از ورودي كاراكتر هاي بعدي خوانده ميشوند تا بهكاراكتري غير از آنچه برروي اين حالت مشخص شده در ورودي برسيد. پس براي تشخيص يك شناسه نيز نياز بخواندن يك كاراكتر اضافي از ورودي هست. در شكل فوق علامت * نمايانگر خواندن يك كاراكتر اضافي تر است . براي مثال در AB12< با ديدن علامت < است كه مي توان فهميد شناسه AB12 به پايان رسيده است. ميتوان شناسه ها را با ماشين خودكار زير نيز نمايش داد:



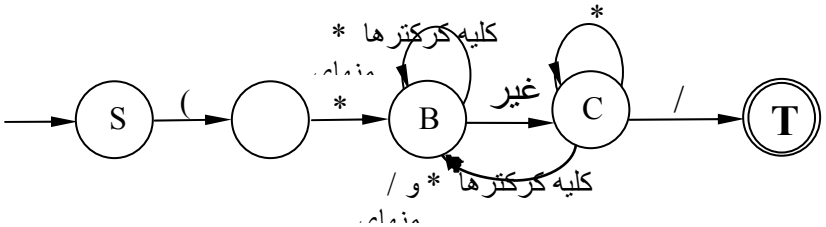
جملات تفسيري Comment توسط تحليلگر لغوي بعنوان يك لغت شناسايي ميشوند. جملات تفسيري پاسكال را ميتوان توسط عبارت زير مشخص نمود :

Comment2 : '(*' ((r | *+ s) * r) * *+ ')'

r : كليه كاراكترها منهاي علامت *

s : كليه كاراكترها منهاي علائم * و ')'

ميتوان جملات تفسيري را با يك ماشين خودكار بصورت زير مشخص نمود:



همانگونه كه مشاهده مي كنيد در حالت شروع S با ديدن كاراكتر / ، گذرى (Transision) از حالت شروع S به حالت **A** وجود دارد و در حالت A اگر * ديده شود گـذري به حالت B وجود دارد. در غير اينصورت اين ماشين خودكار لغت خوانده شده را نمي پذيرد و در اصطلاح Reject مي كند . در حالت B با ديدن هركاراكتري غير از * ماشين خودكار به كار خواندن كاراكترهاي بعدي ادامه ميدهد. اما ورودي * اهمّيت ديگري دارد . با ديدن * اگر در پشت آن علامت / ظاهر شود كار خاتمه يافته تلقي مي شود .

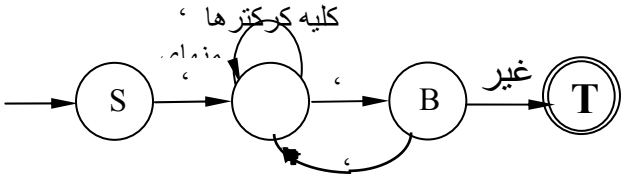
حالت **T** ، حالت پذيرش است . در حالت C در واقع ماشين بخاطر دارد كه * ديده شده است . اگر در حالت C در ورودي علامت / ظاهر شود ، كار خاتمه يافته است. در غير اينصورت ، به حالت B تغيير وضعيت داده ميشود. پس مشاهده مي كنيد كه ماشينهاي خودكار بسيار ساده تر از عبارات با قاعده قابل توليد هستند

فرم كلي رشته هارا در زبان پاسكال با عبارت با قاعده

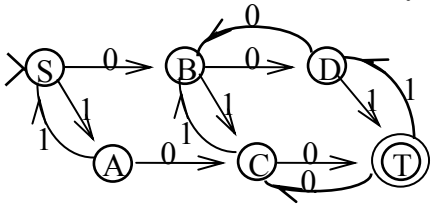
String : ‘ (c | ‘’)* ‘

كليه‌كاراكترها منهاي علامت كوتيشن ' : C

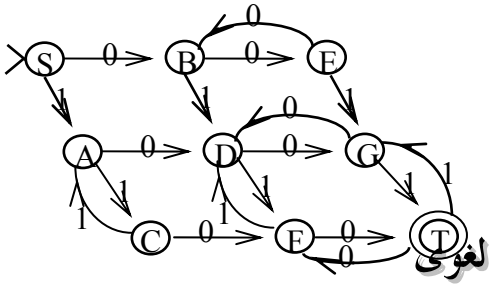
مشخص ميشود. ميتوان رشته ها را با ماشين خودكار زير نيز معين نمود:



مثال: يك ماشين خودكار قطعي براي كليه اعداد مبناي دوكه تعداد صفرهاي آنها زوج و تعداد يك ها فرد است ايجاد كنيد. در ضمن لااقل دو عدد صفر و يك عدد يك بايد در اين اعداد موجود باشد.

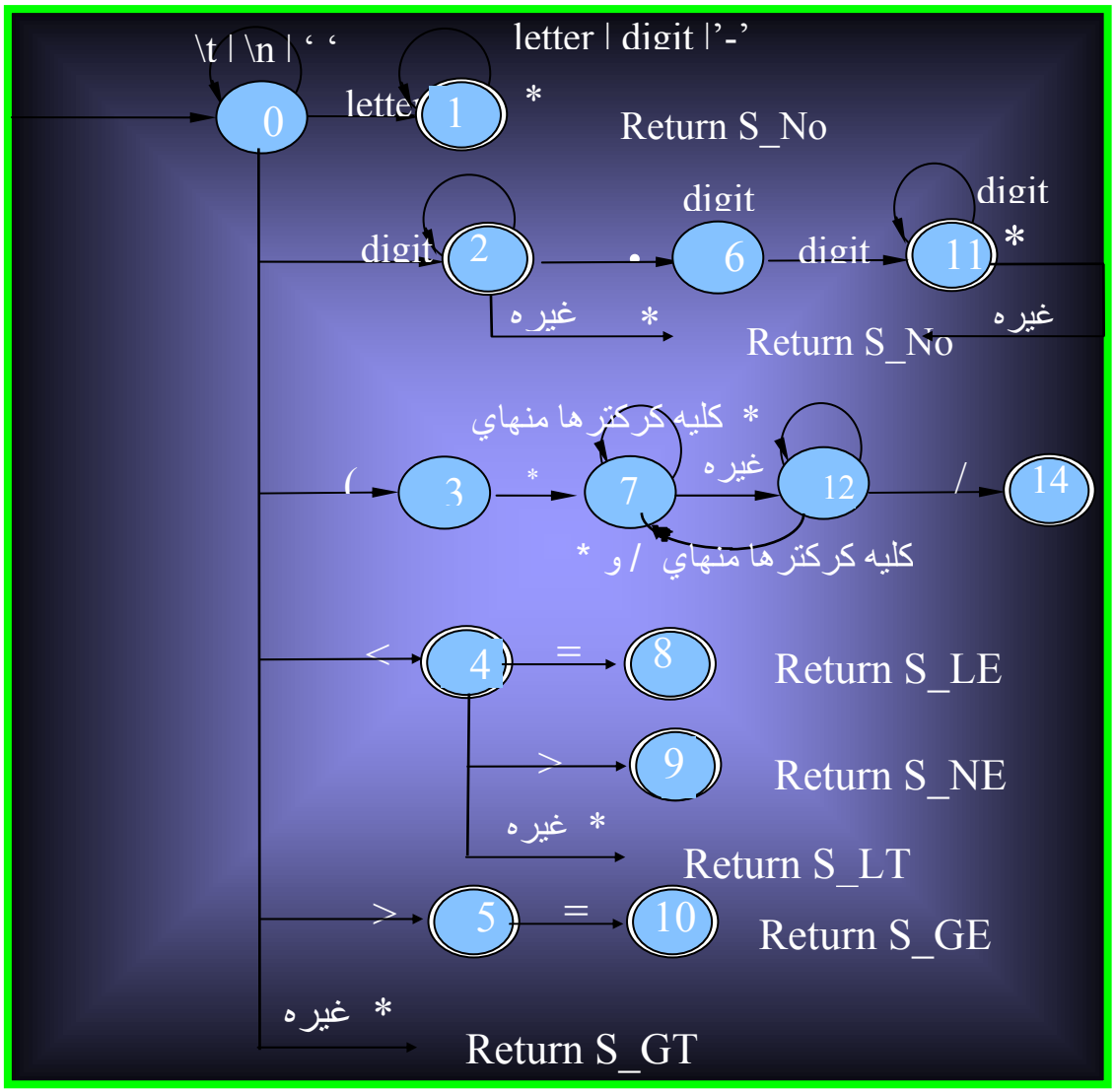


مثال: يك ماشين خودكار قطعي براي كليه اعداد مبناي دوكه تعداد صفرهاي آنها زوج و تعداد يك ها زوج است ايجاد كنيد. در ضمن لااقل دو عدد صفر و دو عدد يك بايد در اين اعداد موجود باشد.



٢.٦ ايجاد تابع تحليلگر لغوى

همانگونه كه در بخش 2و5 ذكر شد ، مي توان با استفاده از ماشينهاي خودكار قوانين لغوي را بيان نمود. در اين بخش نشان داده خواهد شدكه چگونه ميتوان ماشين خودكار را بسادگی تبديل به كد برنامه نمود . براي اين منظور يك دياگرام كلي براي نمايش برخي از لغات ارا ئه مي شود . دياگرام ارائه شده در شكل 2.8 نمايانگر ماشين خودكار كلي براي تشخيص تعدادي از لغات است. در واقع اين دياگرام بيان كننده الگوريتم تابع تحليلگرلغوي است.



شكل 2.8 - دياگرام تابع تحليلگر لغوي

هرگاه كه تابع تحليلگر لغوي مورد فراخواني واقع مي گردد ، در حالت شروع صفر قرار مي گيرد . اگر در فراخواني قبل، كاراكتري اضافه خوانده شده بود ، آن كاراكتر اضافه را مورد استفاده قرار مي دهد ، وگرنه كاراكتر بعدي را از داخل متن برنامه مورد كامپايل مي خواند. تا زماني كه كاراكترهايي كه ارزش لغوي ندارند مانند Tab ، Blank و Newline خوانده شوند ، تحليلگر لغوي در همان حالت شروع صفر باقي ميماند. در غير اينصورت وابسته به نوع كاراكتر ، در يك جمله **Case** اقدام به تشخيص لغات مختلف مي نمايد .

ميتوان با استفاده از دياگرام شكل 2.8 كد تابع تحليلگر لغوي را بشرح زير در زبان C ايجاد نمود.

```
struct TokenType  lexer ( FILE   *InText
                                                    )
{
   enum   Symbols    LexiconType  ;
   char   NextChar  ;   NextWord[80 ]  ;
   int   State , Lentgh  ;
static char LastChar = '\0';
static int RowNo =0, ColNo = 0;
     State   =  0  ;    //  در وضعيت شروع قرار مي‌گيرد
   Length  =  0 ;
   while  ( ! feof  (  InText  ) )
    {
      if  ( LastChar  )
         {  Nextchar  =  LastChar   ;   LastChar  =  '\0'  ;  }
      else
        NextChar   =     fgetc  (  InText  ) ;
       NextWord[Length++] = NextChar;
       switch   State
          {
        case 0:                  //      حالت شروع صفر
              if  (NextChar == '\n') {RowNo++ ;  ColNo =  0;  }
                                        else  ColNo++;
              if  ( Nextchar  = = '  '  ||  Nextchar = = ' \ t '  ||
                   Nextchar  = =  ' \ n '  )  Length = 0;
              else if  (( Nextchar < =  ' z '  & &
                          Nextchar  > = ' a '  )  ||
                         ( Nextchar  < = ' Z '  & &
                          Nextchar  > = ' A '  )      State  = 1;
               else  if  ( Nextchar  < =  ' 9 '  & & Nextchar  > =  ' 0 '  )
                                                  State = 2 ;
               else if  ( Nextchar  = =  ' ( '  )        State 3;
```

```
            else if ( Nextchar  = =  ‘ < ‘ )        State  4
            else if ( Nextchar  = =  ‘ > ‘ )        State  5
           else    LexerError(NextWord, Length);
    break  ; //   پايان حالت صفر
        case    1 :   //   تشخيص شناسه ها
            if( Isalpha( Nextchar ) || Isdigit( Nextchar ) ||
                          NextChar == ‘_’)  State = 1;
          else  { Lastchar  = Nextchar ;
                  NextWord[Length-2] = ‘\0’;
                  return MakeToken(IsKeyWord(NextWord)); }

           Break  ;
    case   3  : // تشخيص اعداد
           .
           .
  } //   پايان سوئچ
} //   پايان حلقه
```

در مثال فوق تابع MakeToken کار ايجاد بسته لغات يا در اصطلاح Token را بر عهده دارد. IsKeyWord کار تشخيص نوع لغات کليدي را بعهده دارد. اين تابع در زبان C بشرح زير است:

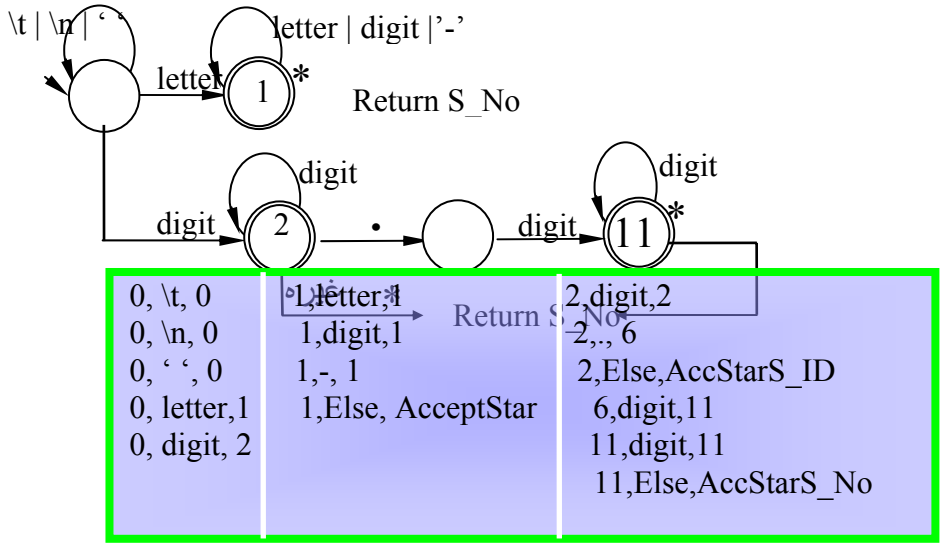
```
enum Symbols IsKeyWord( char *key)
  {
    int I ;
    struct  KeyType
        { char *key;
          enum Symbols Type
        } KeyTab[] = { “if”, S_If,
                       “while”, S_While,
                        “then”, S_Then,
                        “else”, S_Else,
                        “integer”, S_Integer,
                        “type”, S_Type,
                        “function”, S_Function,
                          0, 0};
    for( I  =0;  KeyTab[ I ].key &&
                 strcmp(KeyTab[I].key, Key); I++) ;
    if(KeyTab[I].key) return KeyTab[I].Type;
    Return S_Identifier;
} // EOF IsKeyWord
```

۲.۷ ايجاد مولد تحليلگر لغوی

ميتوان به سادگی تابع مولد تحليلگر لغوي ايجاد نمود. كافيست ، تابعي براي پيمايش يك گراف بنويسيد. اين تابع در ورودي خود علاوه بر متن برنامه مورد تحليل لغوي ، فايل قوانين لغوي را در فرم يك ماشين خودكارمي پذيرد. اين تابع با خواندن هركاراكتراز داخل فايل متن برنامه مورد تحليل لغوي به پيمايش ماشين خودكار ميپردازد تا نوع لغت را تشخيص دهد.

از مولدهاي شناخته شده تحليلگر لغوي ، يكي **LEX** ميباشد. اين مولد امروزه بخصوص بر روي سيستمهاي عامل **UNIX** موجود مي باشد. كار با **LEX** مشكل است چرا كه نياز به بيان قوانين لغوي ، در فرم عبارات با قاعده دارد. در صورتي كه همانگونه كه تا كنون مشاهده كرده ايد، به فرمي خيلي ساده تر مي توان با استفاده از ماشينهاي خودكار، قوانين لغوي را بيان نمود .

بايد قوانين لغوي را در قالب يك ماتريس مشخص نموده و در داخل يك فايل متن قرار داد. سپس در داخل برنامه ميتوان به اين فايل متن ارجاع و ماشين خودكار را در قالب يك ماتريس مشخص نمود. براي نمونه در زير قوانين لغوي در قالب ماشين خودكار و ماتريسي مشخص شده است.



براي ايجاد ماتريس نياز به يك زباني ساده است كه براساس آن براي نمونه بتوان مشخص نمود كه براي مثال Letter ئيست و يا **غيره** ، كه مي باشد . شايد بهتر بود كه در فايل ورودي در داخل ماتريس قوانين لغوي Letter بصورت زير مشخص ميشد:

0 ; [A - Z , a - z] ; 1

حالا در داخل برنامه ، اين ساختار را به عنوان يك فايل **TEXT** ،كاراكتر به كاراكتر خوانده ، عينا در داخل يك ماتريس با تعداد سطرها كه مساوي با تعداد ركوردهاي فايل است وتعداد ستونها كه مساوي با طول هر ركورد يا هر سطر است ، ضبط نمود.
اكنون يا با نوشتن يك برنامه پيمايش كننده ماشين خودكار و يا توليد متن يك برنامه **C** كه بر اساس اين ماشين خودكار براي انجام عمل تحليل لغوي ايجاد شده ، مي توان لغات را تشخيص داد. همزمان با خواندن هركاراكتر از ورودي ، مثلا با خواندن كاراكتر **كوچكتر** از ورودي طبق ماتريس از حالت شروع صفر به حالت **5** تغيير وضعيت داده مي شود ء در اينجا سرعت عمليات تحليلگر لغوي ، وابسته به سرعت جستجو در داخل ماتريس است .
مي توان با ايجاد يك تحليلگر لغوي براي تشخيص متن برنامه ها با لغات كليدي فارسي اقدام نمود. براي مثال جمله زير را ميتوان به يك جمله if تبديل نمود.

حقوق > 1000

اگر

مالیات = حقوق / 100 : آنگاه

۲.۸ تبديل برنامه ها به زبان فارسي

ميتوان متن برنامه هاي فارسي را كه بر اساس قواعد زبانهاي متفاوت نوشته شده است را با استفاده از يك جدول تبديل كلي به زبان انگليسي تبديل و بالعكس پس از كامپايل و اشكال زدايي برنامه ها دوباره با استفاده از همان جدول از انگليسی به فارسي تبديل نمود .

اصولا زبانهاي سطح بالا به خاطر خوانا بودنشان ، **سطح بالا** خوانده مي شوند. متاسفانه اين انگليسی بودن زبانها، خواناييِ برنامه ها را با مشكل مواجه نموده است . بخصوص در انتهاي مراحل تجزيه وتحليل سيستم ها، برنامه سازان را در تبديل متن عمليات تحليل شده از شبه دستورالعمل يا در اصطلاح **Pseudo code** به كد برنامه عملاً با مشكل زياد روبرو نموده است. شبه دستورالعمل را فارسي زبانها نمي توانند به سادگی به زبان انگليسی بنويسند. شبه دستورالعمل ها معمولاً از زبانهاي سطح بالا الهام گرفته مي شوند. مشكل زبان فارسي از راست به چپ بودن جملات وبالعكس ، چپ به راست بودن عبارات است . البته ، در اينجا اين مشكل موردي ندارد.

در مورد شناسه ها نيز در زبان فارسي مشكل وجود دارد . شناسه ها از انتها تشخيص داده مي شوند . مشكل ديگر اين است كه در زبان انگليسی 28 و در زبان فارسي 32 حرف وجود دارد . البته ، با تركيب حروف مي توان اين مشكل را حل نمود .

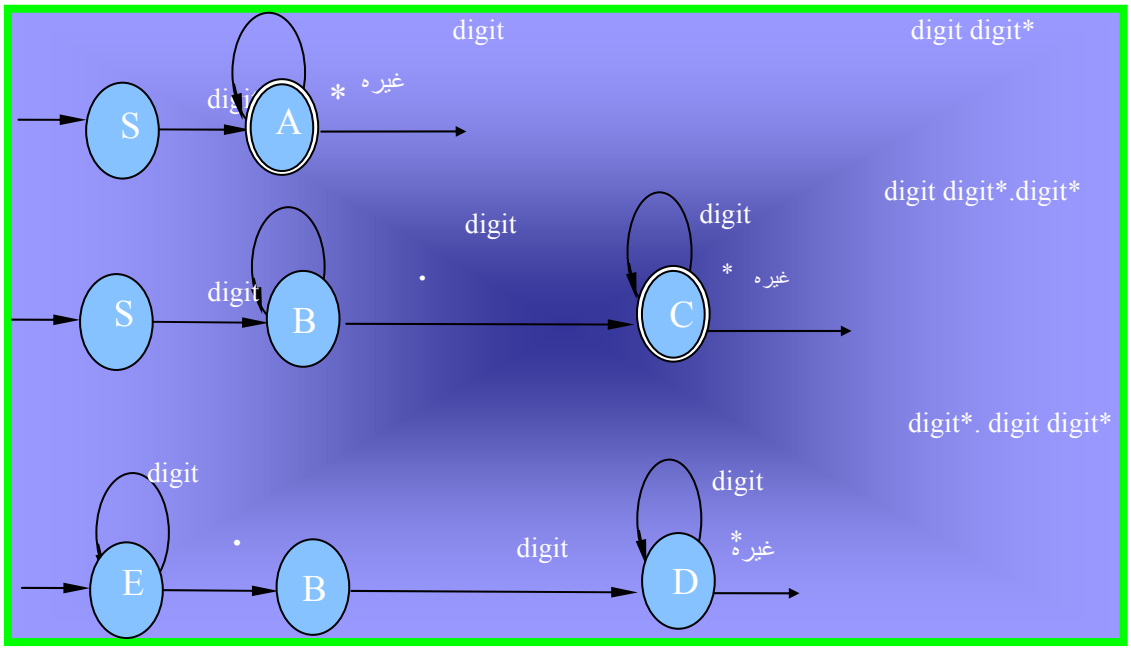
نكته ديگر، محيط ويرايش متن برنامه هاست . دراغلب محيط هاي ارائه شده براي كامپايلرها، **Syntax Directed Editors** ويرايشگرهايي مي باشند كه بنابر قوانين زبان به برنامه ساز كمك مي كنند تا بتواند برنامه خود را با Syntax صحيح ايجاد كند . ميتوان يك محيط قوي ويرايش وابسته به زبان مورد نظر براي ويرايش برنامه هاي فارسي ايجاد نمود.

به اين ترتيب ميتوان برنامه ها را به زبان فارسي با استفاده از يك ويرايشگر باهوش براي زبان خاص نوشت . سپس با استفاده از جدول تبديل برنامه به زبان اصلي ترجمه و پس از اشكالزدايي دوباره از زبان اصلي با استفاده از جدول تبديل لغات را به فارسي برگرداند.

۲.۹ ماشينهاى خودكار غير قطعى

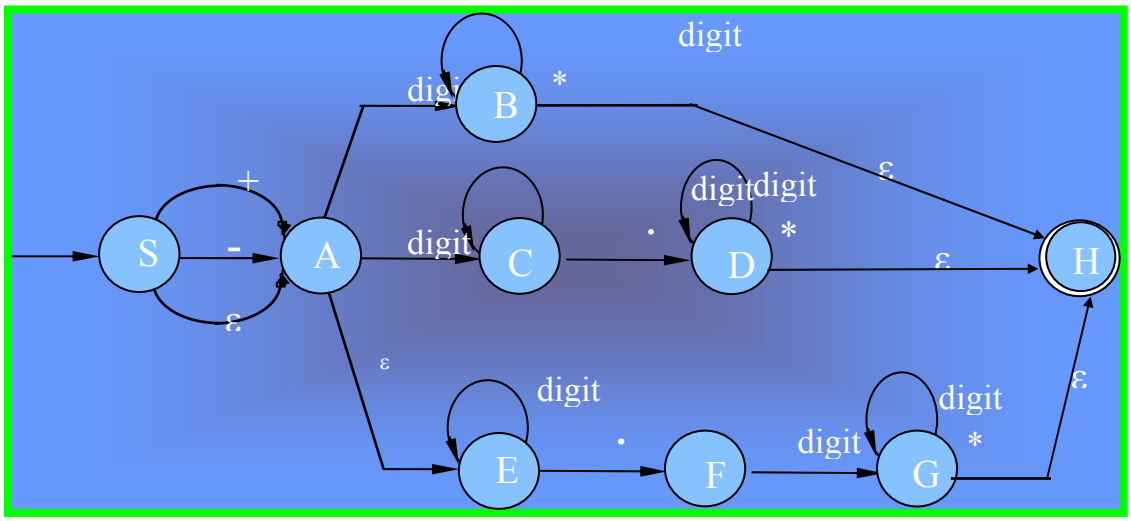
اصولا يك ماشين خودكار در واقع ابزاري براي نمايش چگونگى تصميم گيريهاست . ماشين خودكار يا بصورت قطعي تصميم گيريها را انجام مي دهد يا غير قطعي. اگر به ماشينهاي خودكاري كه تا به حال ترسيم شده توجه كنيم ، مشخص است كه در هر حالتي بطور قطعي مشخص ميكنند كه به ازاي ديدن يك رشته در ورودي از يك حالت به كدام حالت مي بايست گذر نمود . ماشينهاي خودكار قطعي را اصطلاحا" Dterermisitic Finite State Automata گويند.

در ماشينهاي غير قطعي يا در اصطلاح NonDeterministic ، عدم قطعيت در تصميمات وجود دارد . البته اين عدم قطعيت اگرچه كه كار تصميم گيري را مشكل ميكند، اما كار ايجاد ماشين را بسيار ساده تر مي نمايد. حالا، نكته در اينجاست كه چگونه ميتوان يك ماشين خودكار غير قطعي را بصورت قطعي تبديل نمود. در واقع زيبايي الگوريتمها ها در اين زمينه تبلور خود را نشان مي دهد. براي اين منظور اعداد را در نظر بگيريد . بطور مجزا مي توان گفت كه در حالت كلي سه فرم براي اعداد صحيح واعشاري مي تواند وجود داشته باشد :



شكل 2.8- انواع اعداد صحيح و اعشاري

در اينجا ترسيم سه ماشين خودكار براي اعداد مستقل از يكديگر ساده است . مشكل ، تركيب اين سه حالت و ايجاد يك ماشين خودكار براي سه نوع عدد مي باشد. با استفاده از ماشينهاي خودكار غير قطعي ميتوان به سادگي و بدون در نظر گرفتن مشكل تركيب گراف براي سه نوع متفاوت اعداد اقدام به ترسيم يك ماشين خودكار براي تشخيص سه نوع عدد نمود. زيبايي كار در اينجاست كه ميتوان بصورت الگوريتمي و بدون نياز به هيچگونه كار اضافي ، اين سه حالت را در قالب يك ماشين خودكار غير قطعي با يكديگر ادغام نمود و با استفاده از الگوريتمهايي كه ارائه خواهد شد بطور اتوماتيك تبديل به يك ماشين خودكار قطعي نمود.



شكل 2.9- ماشين خودكار غير قطعي

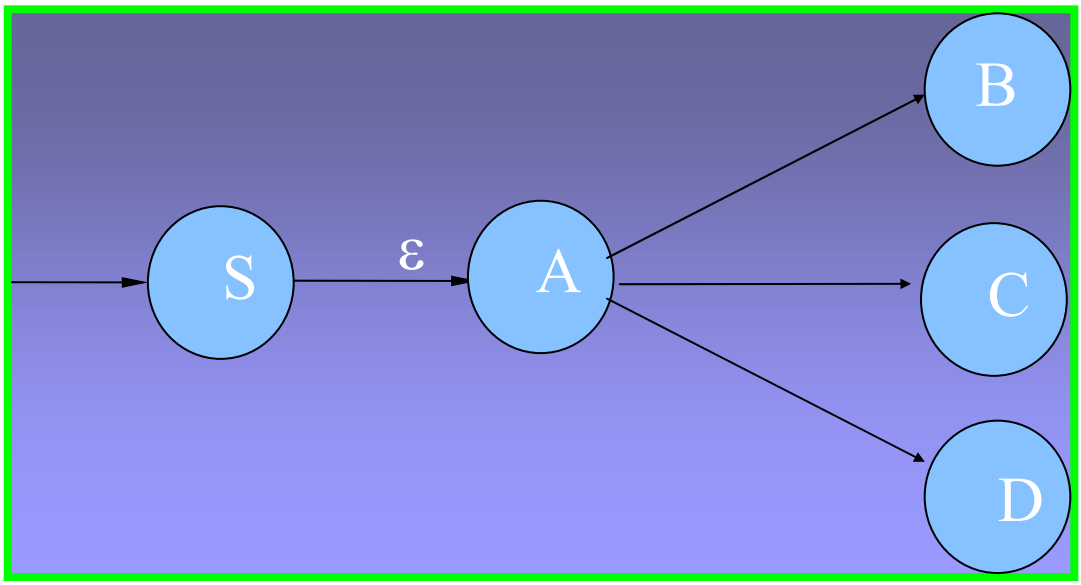
همانگونه كه در شكل 2.9 مشاهده مي شود ، يك ماشين خودكار غير قطعي مي تواند بيان گر حالات مختلف براي يك لغت باشد. در اينجا منظور از لغت ، اعداد مي باشند . اعداد يا داراي علامت هستند و يا علامت ندارند. اين نداشتن علامت را با گذر تهي كه با علامت ε (اپسيلون) مشخص شده ، معين مي كنند. همانگونه كه مشاهده مي شود، در حالت A با ديدن digit نمي توان مشخص نمود كه حالت بعدي آيا حالت B يا C ويا حالت E است . بايد توجه داشته باشيد كه گذر ε در واقع يعني هيچ چيز .

براي تبديل ماشين خودكار غير قطعي به قطعي در سه مرحله عمل ميشود:

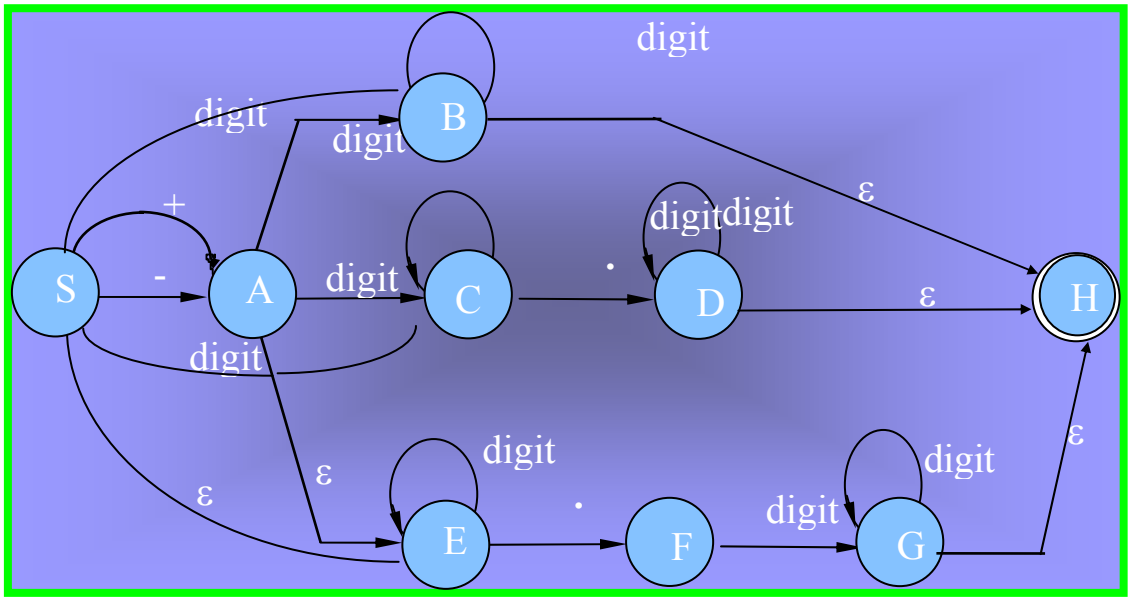
1. حذف گذرهاى تهي
2. رفع عدم قطعيت
3. بهينه سازي ماشين خودكار

۲.۹.۱ حذف گذرهای تهی

اگر از حالت S گذری تهي به حالت A وجود دارد، اين نمايانگر يك تساوي يك طرفه مي باشد كه نكته ايست جالب توجه . در اينجا S با A هيچ تفاوتي ندارد اما A متمايز از S است. ميتوان هر واكنشي كه در A وجود دارد را براي S نيز در نظر گرفت اما بالعكس صادق نيست.

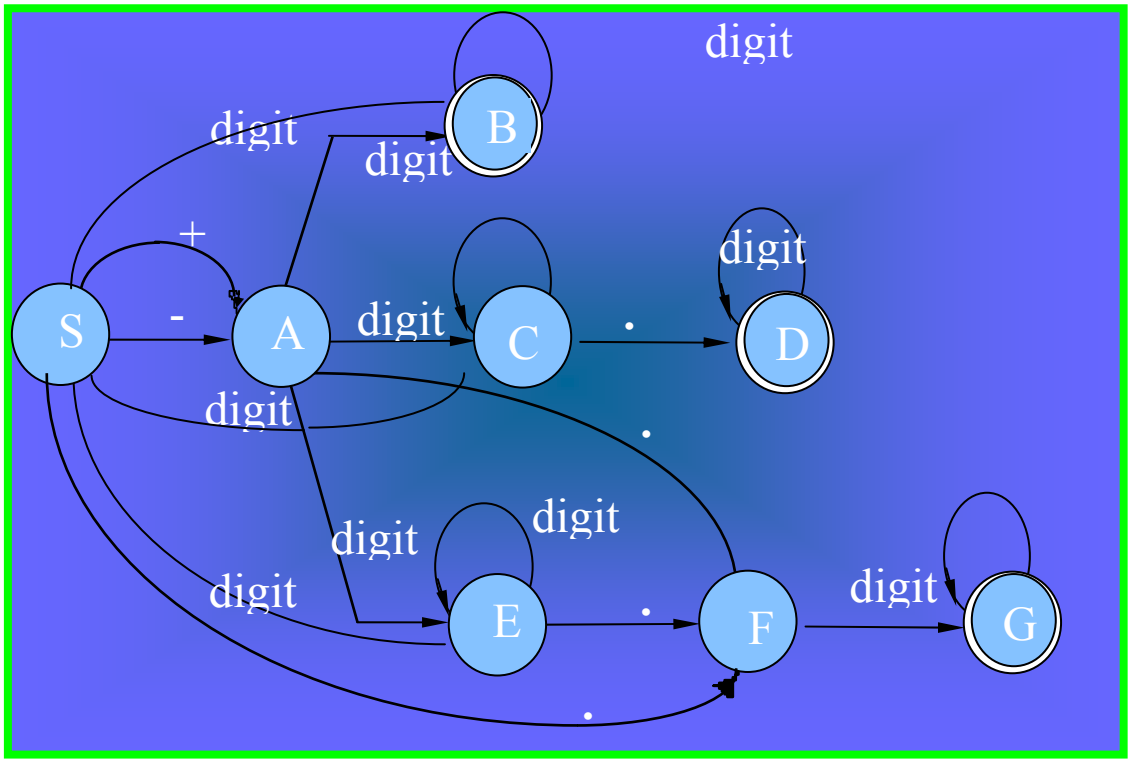


براي درك بهتر مطلب ، گذر تهي از حالت صفر به حالت يك در شكل 2.9 را در نظر بگيريد اگر عدد علامت نداشته باشد، حالت صفر دقيقاً مثل حالت يك است. اما، اگر عدد علامت داشته باشد ، حالت صفر واكنشي ديگر دارد كه واكنش اين حالت ، گـذري به حالت يك مي باشد. حالت صفر از حالت يك متمايز گرديد ، چرا كه ممكن بود عدد داراي علامت باشد، اما اگر عدد علامتي نداشته باشد اين دو حالت يكي خواهند بود. پس هيچگونه نيازي به جداسازي آنها نخواهد بود .



شكل 2.10 حذف گذر تهي از حالت صفر به يك

همانگونه كه مشاهده مي شود، كليه واكنشهاي حالت يك و يا به عبارت ديگر كليه گذرهاي خارج شونده از حالت يك ، براي حالت صفر در نظر گرفته شد . به اين ترتيب گذر تهي حذف گرديد. در مرحله بعد به همين ترتيب ادامه مي دهيم تا كليه گذرهاي تهي را از داخل ماشين خودكار قطع نما ييم .

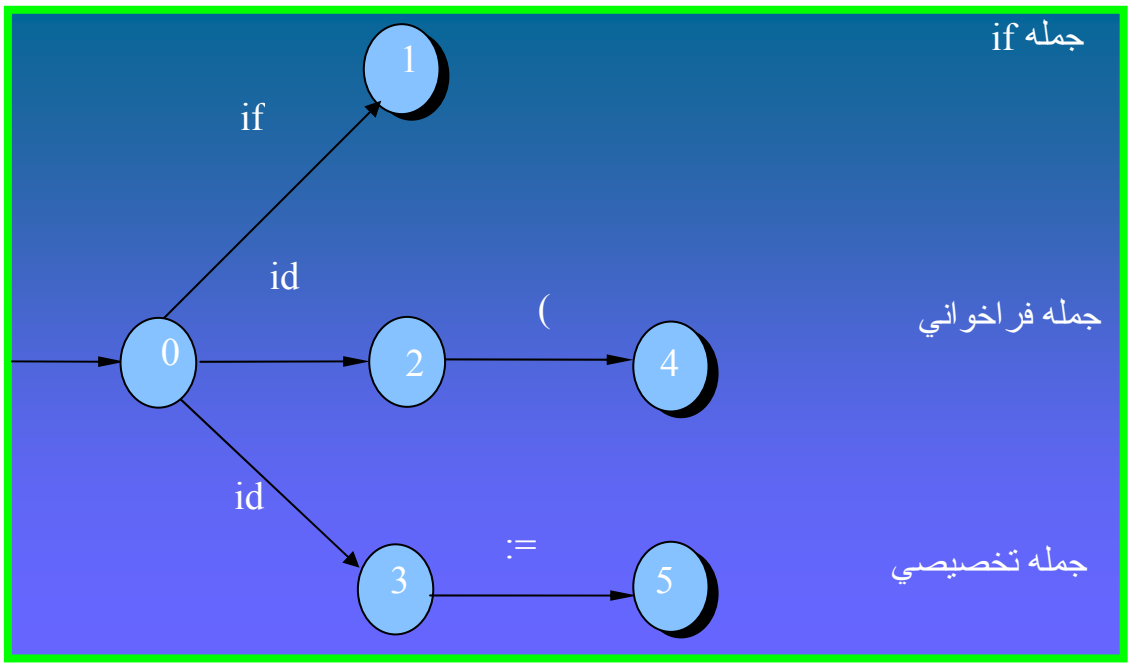


شكل 2.11- ماشين خودكار بدون گذرهاي تهي

۲.۹.۲ رفع عدم قطعيت

همانگونه كه در شكل **2.11** مشاهده مي كنيد، ماشين خودكار غير قطعي است. براي نمونه در حالت A با ديدن digit نمي توان تشخيص داد كه آيا حالت بعدي **B** ، **C** و يا اينكه **E** مي باشد. اما نكته جالب توجه اينجاست كه با ديدن digit حتماً به يكي از اين **سه** حالت گذر مي شود . براي روشن تر شدن روش حذف عدم قطعيت بهتر است كه كار تحليلگر نحوي مطرح شود .

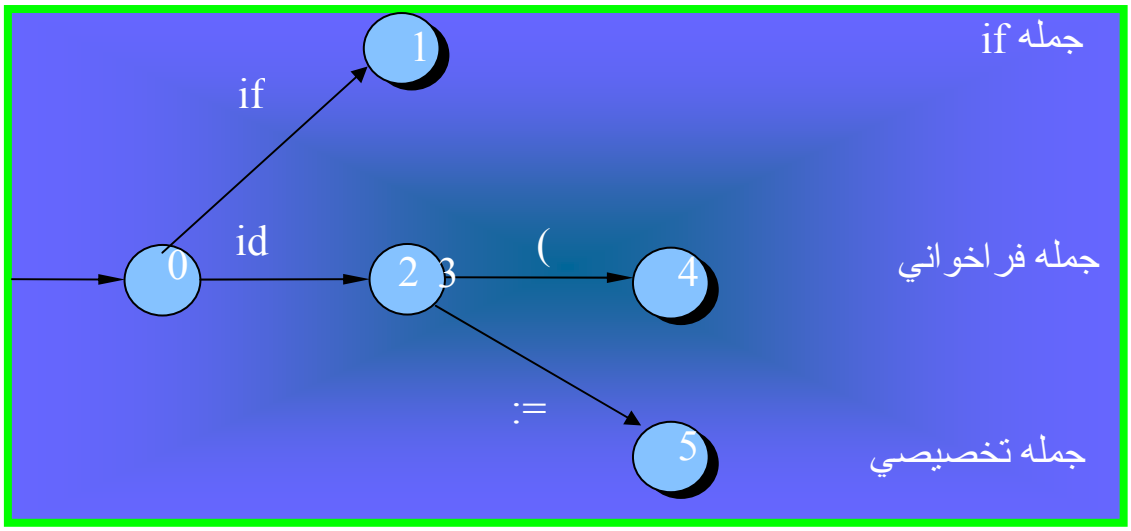
تحليلگر نحوي ، معمولاً با ديدن يك لغت در ورودي ، ساختار جمله مورد نظر را مي تواند پيش بيني كند. براي مثال اگر لغت دريافتي از تحليلگر لغوي **if** باشد، تحليلگر نحوي بلافاصله تصميم مي گيرد كه جمله بايد جمله **if** باشد و بايد انتظار ديدن شرط جمله **if** را در ورودي داشته باشد. اما، با ديدن يك اسم ، يا به عبارت ديگر يك شناسه ، مشكل وجود دارد. در اينجا تحليلگر نحوي نمي تواند مشخص كند كه آيا جمله مورد نظر يك جمله **فراخواني** زير برنامه ها ويا يك جمله **تخصيصي** (**Assignment**) است. ماشين خودكار غير قطعي در اينجا بصورت زير قابل ترسيم است :



شكل 2.12 تشخيص جملات if ، فراخواني و تخصيصي

عليرغم وجود عدم قطعيت در حالت صفر از شكل 2.12 ، مي توان قطعيتي را در نظر داشت. به اين ترتيب كه در حالت شروع صفر اگر يك شناسه يا id در

ورودي ظاهر شود، قطعاً حالت بعدي ، حالت 2 يا 3 خواهد بود. در حالت 2 يا 3 اگر ':=' در ورودي ظاهرشود جمله تخصيصي ، اگر '(' ظاهر شود جمله فراخواني و در غير اينصورت جمله غلط مي باشد. پس حالات 2 يا 3 را بصورت يك حالت تركيبي جديد با ادغام خروجي هاي اين دو حالت مي توان بوجود آورد :



شكل 2.13 ماشين خودكار قطعي

براي سهولت در امر تبديل ويا نمايش ماشينهاي خودكار به جاي گراف از فرم جدول مانند استفاده ميشود. براي نمونه ماشين خودكارغير قطعي اعداد در شكل **2.11** ، بصورت يك جدول در شكل 2.14 مشخص شده است.

	digit	.	+/-
S	B,C,E	F	A
A	B,C,E	F	
(B)	B		
C	C	D	
(D)	D		
E	E	F	
F	G		
(G)	G		

شكل 2.14 جدول ماشين خودكار اعداد

همانگونه كه در شكل 2.14 مشاهده ميشود در حالات S و A به يكي از سه حالت **B** يا **C** يا **E** به ازاي ديدن digit گذري وجود دارد. لذا ، ميتوان گفت كه گذري به حالت تركيبي BCE وجود دارد. در اين حالت ميتوان واكنشهاي هر سه حالت B ، C و E را داشت. رديف مربوط به حالت تركيبي BCE در شكل 2.15 ، تركيبي از واكنشهاي هر سه حالت تشكيل دهنده آن است به عبارت ديگر از تركيب "يا"

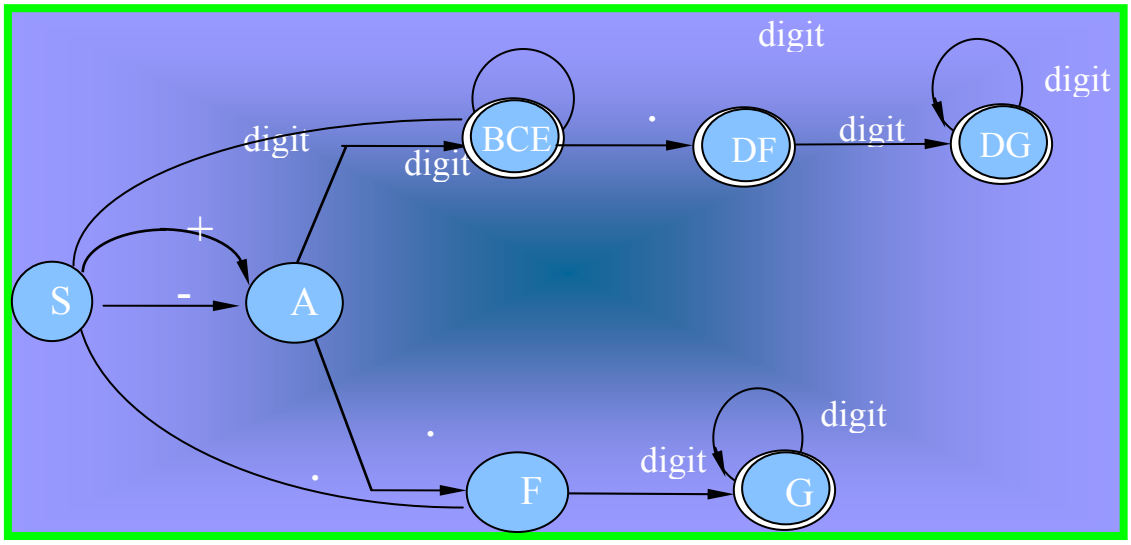
سطرهاي B ، C و E، سطر جديد BCE به ماتريس افزوده مي شود . BCE يك حالت پذيرشي است ، زيرا يكي از حالات تشكيل دهنده آن يعني B يك حالت پذيرش است .

همانگونه كه در شكل 2.15 مشاهده ميشود ، در حالت جديد BCE به ازاء ورودي '.' (نقطه اعشار) عدم قطعيت وجود دارد . از حالت BCE ميتوان به هر يك از دو حالت D يا F گذر نمود. لذا براي رفع عدم قطعيت، حالت جديد ديگري بنام DF را به جدول حالات بايد افزود. براي رفع عدم قطعيت در حالت DF حالت DG ايجاد ميشود.

	digit	.	+/-
S	BCE	F	A
A	BCE	F	
(B)	B		
C	C	D	
(D)	D		
E	E	F	
F	G		
(G)	G		
(BCE)	BCE	DF	
(DF)	DG		
(DG)	DG		

شكل 2.15 جدول ماشين خودكار اعداد

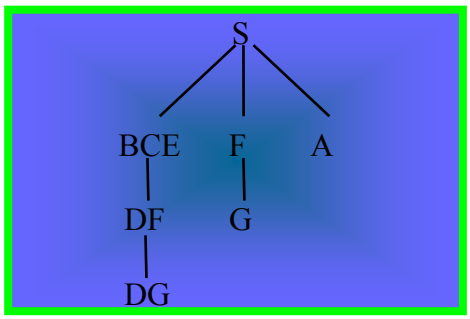
اكنون با استفاده از جدول فوق ميتوان ماشين خودكار قطعي اعداد را بصورت زير ايجاد نمود.



شكل 2.16 ماشين خودكار قطعي اعداد

همانگونه كه مشاهده ميكنيد برخي از حالات درون جدول مثل حالات B ، C ، D و E در عمل غير قابل دسترسي هستند. علت افزايش حالات جديد تركيبي است. ميتوان اين حالات زااِد را با ايجاد درخت دسترسي نيز مشخص نمود.

با افزايش حالات جديد جهت حذف عدم قطعيت در ماشين خودكار ، برخي از حالات زائد شده ، از حالت شروع غير قابل دسترسي ميشوند . جهت تشخيص اين حالات يا مي توان ماشين خودكار را از روي جدول ترسيم نمود و يا اينكه با استفاده از يك درخت دسترسي حالات قابل دسترسي از حالت شروع S را مشخص نمود. براي نمونه درخت دسترسي براي ماتريس شكل 2.15 بصورت زير است. بايد توجه داشته باشيد كه نام حالات در درخت دسترسي تكراري نيست و اين درخت صرفا"جهت تعيين حالات قابل دسترسي ازحالت شروع S است.



شكل 2.17 درخت حالات قابل دسترسي

۲.۹.۳ بهينه سازى ماشينهاى خودكار

هدف از بهينه سازي ، تقليل تعداد حالات در ماشينهاي خودكار است . براي اين منظور بايد حالات معادل را تشخيص داد . دو حالت را در صورتي معادل گويند كه به ازاي ورورديهاي متفاوت با گذشت از يك سري از حالات يا به عبارت ديگر در مسيري به طول صفر يا بيشتر و با گـذشتن از تعداد **n** گره نهايتا" به حالت پذيرش برسند. حالات معادل را با روشي ابتكاري به ترتيب زير مي توان مشخص نمود :

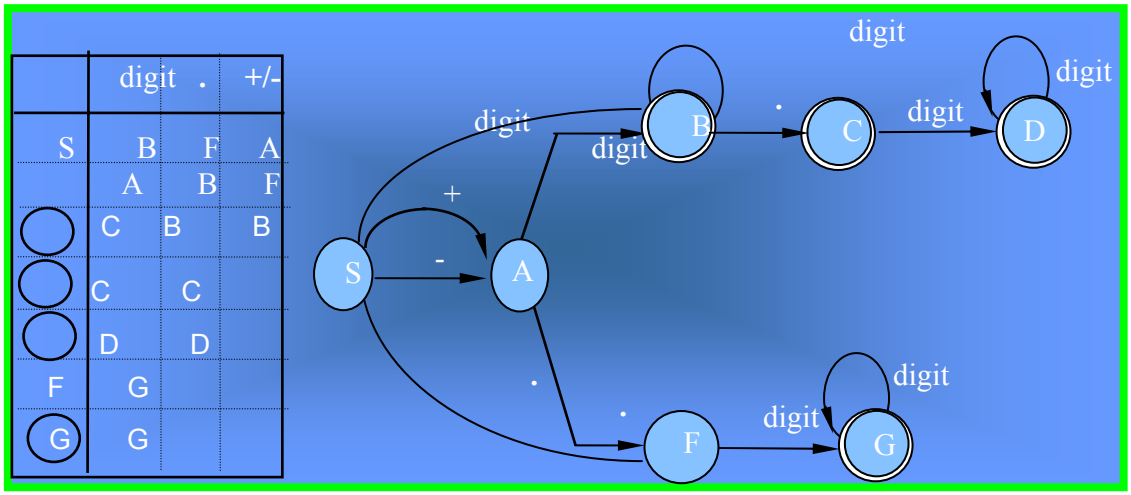
1. ابتدا حالات معادل صفر كه با مسيرهائـي به طول صفر به پذيرش ميرسند را از ساير حالات مجزا بايد نمود. بعبارت ديگر ، حالات پذيرش از حالات غير پذيرش مجزا مي شوند. به اين ترتيب حالات به دو دسته مجزا از حالات پذيرش و غير پذيرش تقسيم ميشوند.
2. به درون هر دسته حالات بايد نگريست ومشخص نمود كه آيا ورودي اي وجود دارد كه به ازاي آن حالات متعلق به يك دسته واكنشهای جدا از سايرين دارند. واكنشها بر اساس گذر يك حالت به حالت ديگر سنجيده نمي شود بلكه ، اگر از يك حالت به حالت ديگر گذري وجود دارد دسته مربوط به آن حالت را مشخص ميكنند. مقصود مقايسه دسته هاإی است كه به آنها گذر ميشود. حالات مشابه به ازا يك ورودي به حالات درون يك دسته گذر مي كنند .
3. آنقدر مر حله **2** تكرار مي شود تا ديگر گذري متفاوت بين دسته ها وجود نداشته باشد ودسته اي جديدتر توليد نشود . براي نمونه اكنون ماشين خودكار كه در مبحث قبل مطرح شد بهينه سازي مي شود.

بطور خلاصه و با در نظر گرفتن الگوريتم فوق مي توان حالات معادل را بصورت زير تعريف نمود :

دو حالت را در صورتي معادل **k** گويند كه به ازاي هر رشته از ورورديها به طول كوچكتر يا مساوي با **k** اگر از يكي از آنها بتوان به پذيرش رسيد ، از ديگري نيز بتوان به ازاء همان رشته به حالت پذيرش رسيد.

دو حالت را معادل گويند اگر و تنها فقط اگر به ازاي هر رشته اي از ورورديهاي متوالي كه موجب رسيدن از آن حالت به يك حالت پذيرش است بتوان از ديگري نيز به پذيرش رسيد.

اكنون با توجه به تعريف حالات معادل و مراحل بهينه سازي ماشينهاي خودكار ، مي توان مبادرت به بهينه سازي ماشين خودكار اعداد كه در مبحث قبل مطرح شد ، نمود. ماشين خودكار قطعي اعداد و جدول توليد آن در شكل 2.18 ارائه شده است.



شكل 2.17- ماشين خودكارقطعي اعداد

براي بهينه سازي ماشينهاي خودكار ابتدا بايد گره ها را به دو دسته پذيرشی و غير پذيرشی افراز نمود . بعبارت ديگر حالات معادل صفركه با مسيرهايي به طول صفر به پذيرش مي رسند را از ساير حالات بايد تفكيك نمود. به اين ترتيب حالات ماشين خودكار اعداد كه در شكل 2.17 ارائه شده ، به دو دسته زير تقسيم ميشوند:

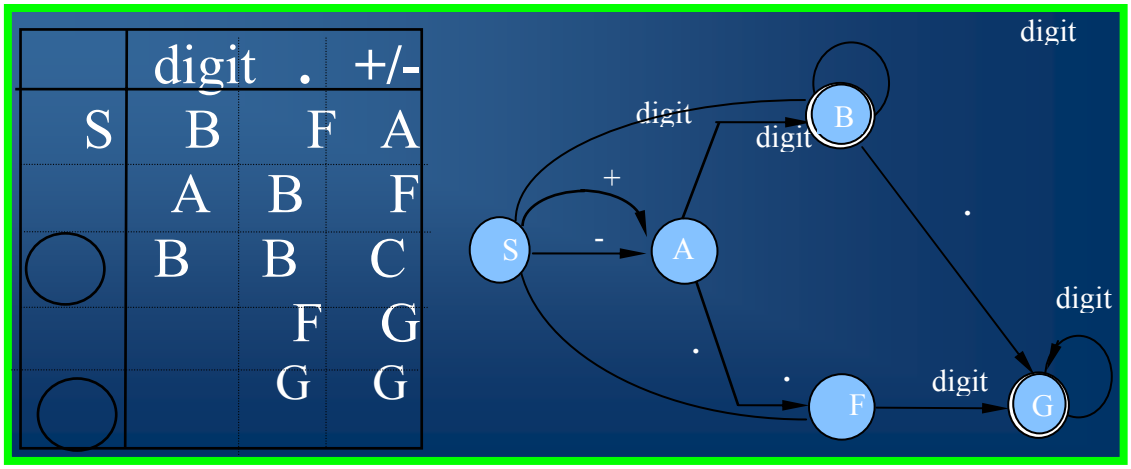
1	{B,C,D,G}	حالات پذيرشی
2	{S,A,F}	حالات غير پذيرشی

حال بايد مشخص نمود كه آيا در داخل هر دسته به ازاء هر ورودي گره ها واكنشي يكسان دارند . منظور از واكنش يكسان اين است كه براي نمونه اگر از حالت M در دسته شماره 12 به ازاء ورودي digit گذري به حالتي در دسته شماره 7 وجود دارد ساير حالات موجود در دسته شماره 12 بايد به ازاء ورودي digit گذري به حالتي در دسته شماره 7 داشته باشند. در غير اينصورت اين حالات از داخل دسته خارج شده و در دسته اي جديد قرار مي‌گيرند. براي نمونه در داخل دسته شماره 1 در بالا به ازاء ورودي نقطه اعشار '.' از حالت B يك خروجي و گذري وجود دارد در صورتيكه در مورد ساير حالات اينچنين نيست. پس اين حالت از سايرين جدا ميشود و سه دسته به صورت زير حاصل مي گردد:

1	{ C ,D ,G}	
2	(B)	
3	{S ,A ,F}	حالات غير پذيرشی

به همين ترتيب در دسته شماره 3 حالت S قابل تفكيك از A و F است. زيرا در S به ازاء ورودیهاي + و - گذر و واكنشي وجود دارد در صورتيكه در مورد A و F

اينچنين نيست. پس حالت S از A و F قابل تفكيك است. از سوي ديگر A و F نيز قابل تفكيك از يكديگر هستند. بنابراين سه حالت C ، D و G غير قابل تفكيك از يكديگر و معادل هستند. پس ميتوان هر يك از اين سه حالت معادل را بجاي دو حالت ديگر در داخل ماشين خودكار قرار داد. به اين ترتيب ماشين خودكار بهينه اعداد بصورت زير خواهد بود.



شكل 2.18- ماشين خودكاربهينه اعداد

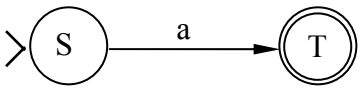
۲.۱۰ تبديل عبارات با قاعده به ماشينهاى خودكار

در اين قسمت نشان داده خواهد شد كه چگونه مي توان عبارات با قاعده را به صورت ماشينهاي خودكار تبديل نمود تا اينكه بتوان با استفاده از ماشين خودكار، قوانين لغوي كه در فرم عبارات باقاعده خلاصه شده اند را عملا" به صورت كد برنامه مورد بهره برداري قرار دادء براي تبديل عبارات با قاعده به ماشينهاي خودكار معمولا" مراحل زير بكار مي روند :

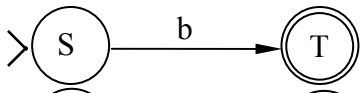
1 . هر عنصر **a** متعلق به الفباي زبان به صورت يك ماشين خودكار نمايش داده مي شود :

2 . ماشين خودكار براي عبارت **ab** را از تركيب گرافها براي **a** و **b** به صورت زير مي توان ايجاد نمود.

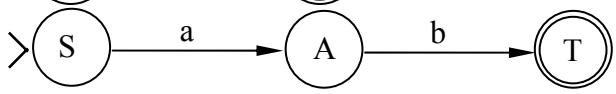
(الف) ماشين خودكار براي a



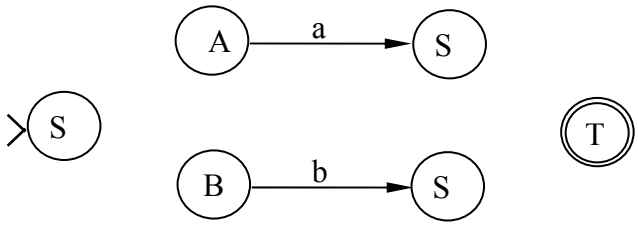
(ب) ماشين خودكار براي b



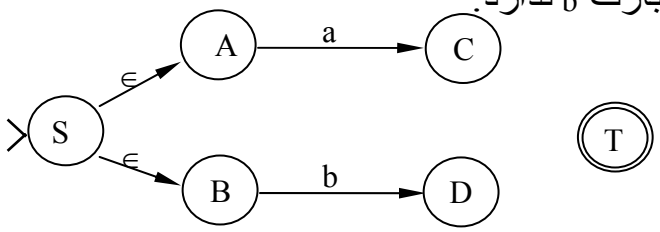
(ج) ماشين خودكار براي ab



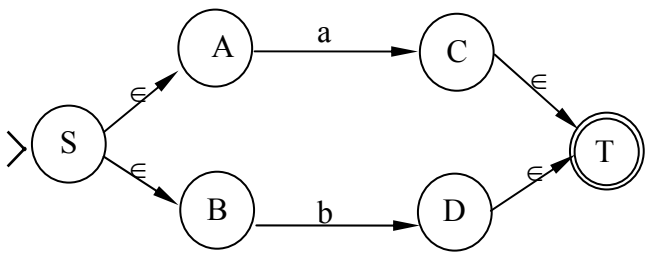
3 . ماشين خودكار براي عبارت **a | b** را با يك استدلال ساده ميتوان از ماشين هاي خودكار براي عبارات a و b ايجاد نمود. چنانچه حالت شروع ماشين خودكار براي عبارت **a | b** حالت S باشد :



حالت شروع S هيچ فرقي با حالت شروع براي عبارت a ندارد زيرا در شروع a | b ميتوان a را هم در ورودي ديد. از سوي ديگر حالت شروع S هيچ تفاوتي با حالت شروع براي عبارت b ندارد.



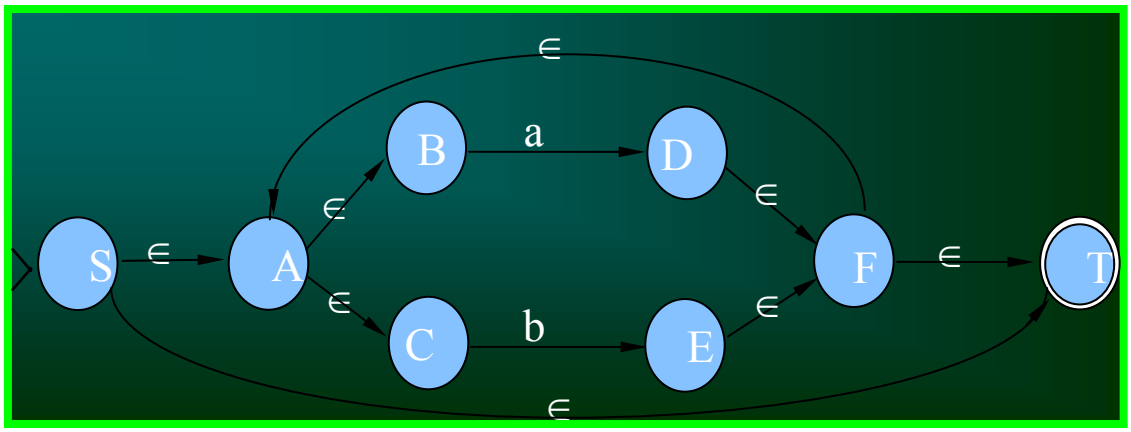
به همين ترتيب با مشاهده a در ورودي بايد مطمئن بود كه a|b در ورودي ظاهر شده است. لذا ، حالت پذيرش براي ماشين خودكار عبارت a يعني C هيئ تفاوتي با حالت پذيرش براي ماشين خودكار براي عبارت a | b يعني T ندارد. به همين ترتيب حالت D هيچ تفاوتي با حالت پذيرش T ندارد. بنابراين ماشين خودكار براي عبارت a | b بصورت زير خواهد بود:



در شكل فوق عبارت **a | b** از تركيب ماشينهاي خودكار براي عبارات **a** و **b** ساخته شده است. مسلم است كه حالت شروع S هيچ تفاوتي با حالت A وحالت B ندارد. اما بالعكس صادق نيست و حالت شروع براي تشخيص a يعني A متفاوت از حالت شروع براي a | b است ء زيرا در حالت شروع براي a نمي توان در ورودي b را ديد. لذا، حالت شروع a | b را با دو گذر تهي به حالت شروع براي a و حالت شروع براي b متصل بايد نمود.

4 . با در دست داشتن ماشين خودكار براي a و b و درنتيجه وجود ماشين خودكار براي a | b مي توان ماشين خودكار براي عبارت (a | b)* را با يك استدلال ساده ايجاد نمود. به اين ترتيب كه فرض كنيد حالات شروع و خاتمه براي ماشين خودكار (a | b)* به ترتيب حالات S و T باشند . اولا" ، چون (a | b)* ميتواند تهي نيز باشد پس حالت شروع آن ممكن است هيچ تفاوتي با حالت پذيرش نداشته باشد. زيرا رشته (a | b)* ممكن است اصلاً وجود نداشته باشد.

از طرف ديگر ممكن است حالت شروع S هيئ تفوتي با حالت شروع (a | b) نداشته باشد زيرا ، در شروع (a | b)* ميتوان در شروع مشاهده a | b در ورودي بود. پس از مشاهده a | b دو باره ميتوان در آغاز مشاهده a | b جديدتري قرار گرفت . اين در واقع بخاطر خاصيت تكراري بودن (a | b)* است.پس ، حالت پذيرش a | b درون (a | b)* ، هيچ تفاوتي با حالت شروع آن نخواهد داشت. با اين استدلال ميتوان نتيجه گرفت كه ماشين خودكار براي عبارت (a | b)* بصورت زير است.

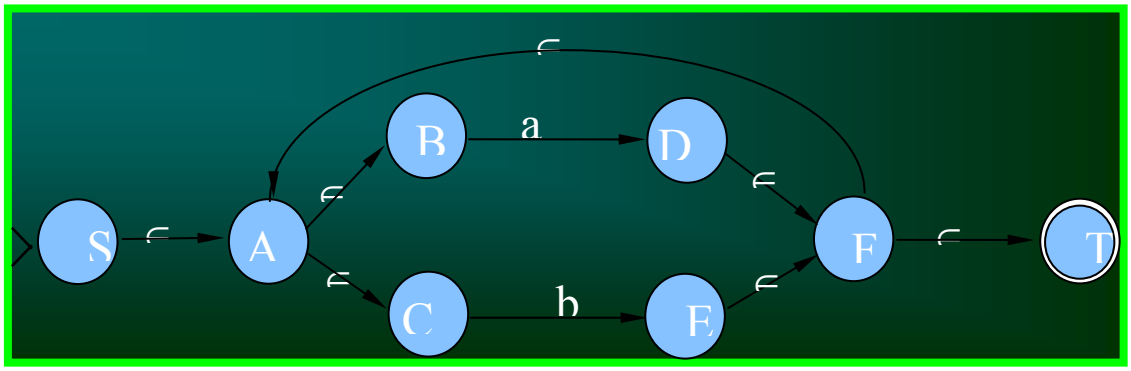


شكل 2.19- ماشين خودكار براي عبارت (a | b)*

مسلما" در ورودي (a | b)* ئون رشته a يا b مي تواند اصلا" وجود نداشته باشد با اين نتيجه حالت شروع مي تواند معادل با حالت پذيرش باشد. لذا ، در شكل 2.19 حالت شروع S با گذري تهي به حالت پذيرش T متصل شده است. از جهت ديگر در خاتمه ديدن a | b مثل اينكه دوباره در حالت شروع بوده و مي توان a | b

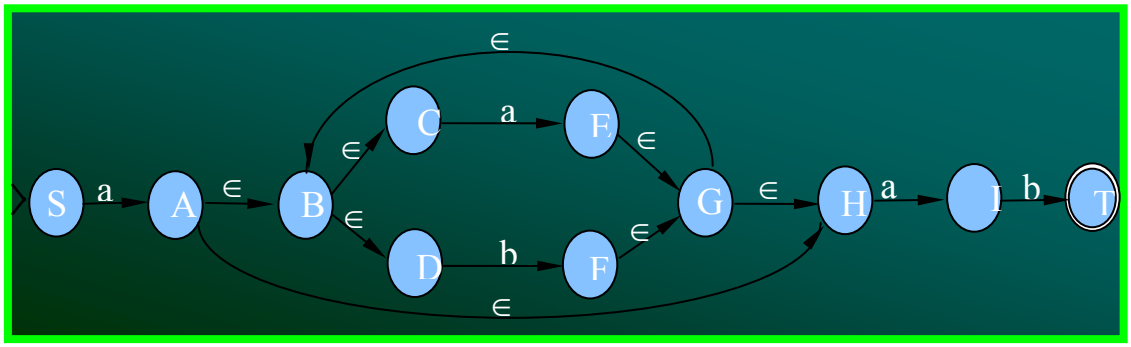
جديدي را مشاهده نمود و اين عمل تا به هر تعدادي قابل تكرار است . براي اين منظور حالت F با گذري تهي به حالت شروع a | b يعني A متصل شده است. علاوه بر اين حالت F ممكن است خاتمه تكرار و حالت پذيرش هم باشد.

5 . عبارت $(a \mid b)^+$ را مي توان با حذف گذر تهي از حالت شروع به حالت پايان در ماشين خودكار براي $(a \mid b)^*$ توليد نمود زيرا درمورد $(a \mid b)^+$ حداقل يكبار a | b در ورودي بايد ظاهر شود و حالت شروع آن با خاتمه يكسان نيست .



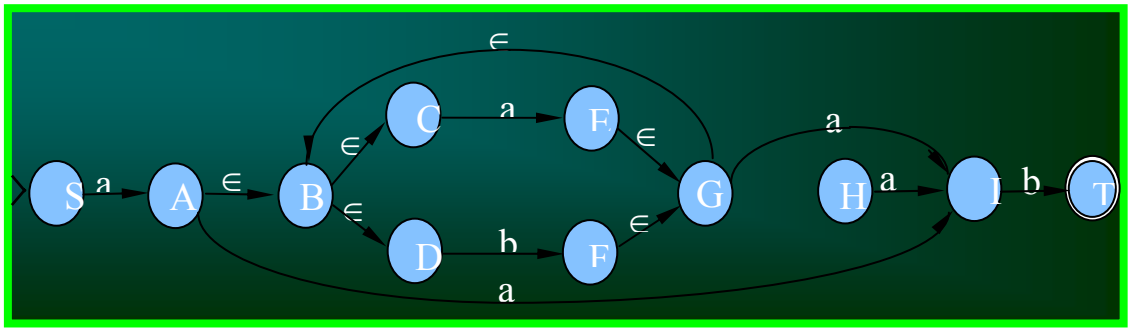
شكل 2.20- ماشين خودكار براي عبارت $(a \mid b)^+$

مثال 1- براي عبارت a (a | b)*a b يك ماشين خودكار بهينه ايجاد نماييد.
با استفاده از ماشين خودكار ارائه شده براي عبارات (a | b)* كه در شكل 2.19 ارائه شده است ، ميتوان بسادگی ماشين خودكار غير قطعي را ايجاد كرد.



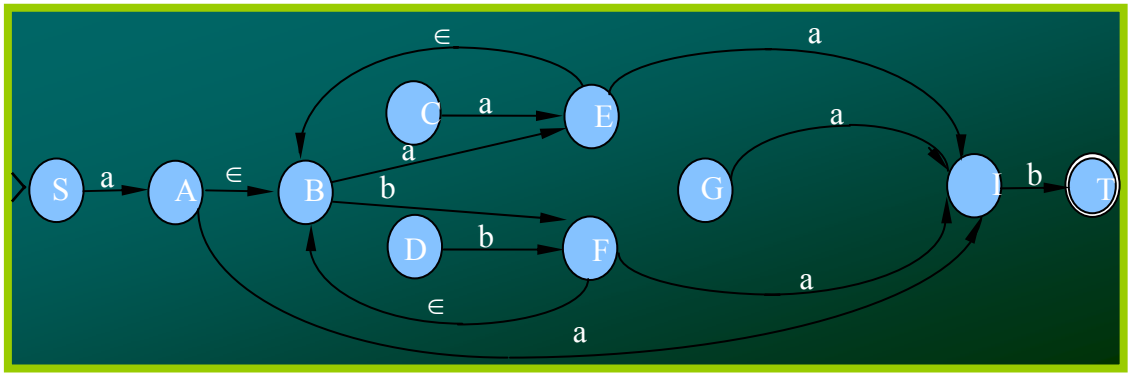
شكل 2.21- ماشين خودكار غير قطعي براي عبارت

پس از حذف گذرهاي تهي به H ماشين بصورت زير تبديل ميشود.



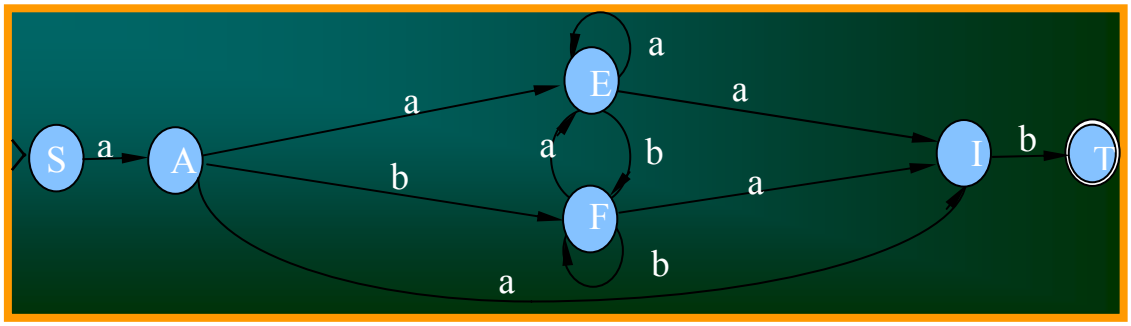
شكل 2.22- حذف گذرهاي تهي از A و G به H

مشاهده ميكنيد كه حالت H در شكل 2.22 غير قابل دسترسي و قابل حذف است. در شكل 2.23 گذرهاي تهي به حالات G ، C و D حذف شده اند.



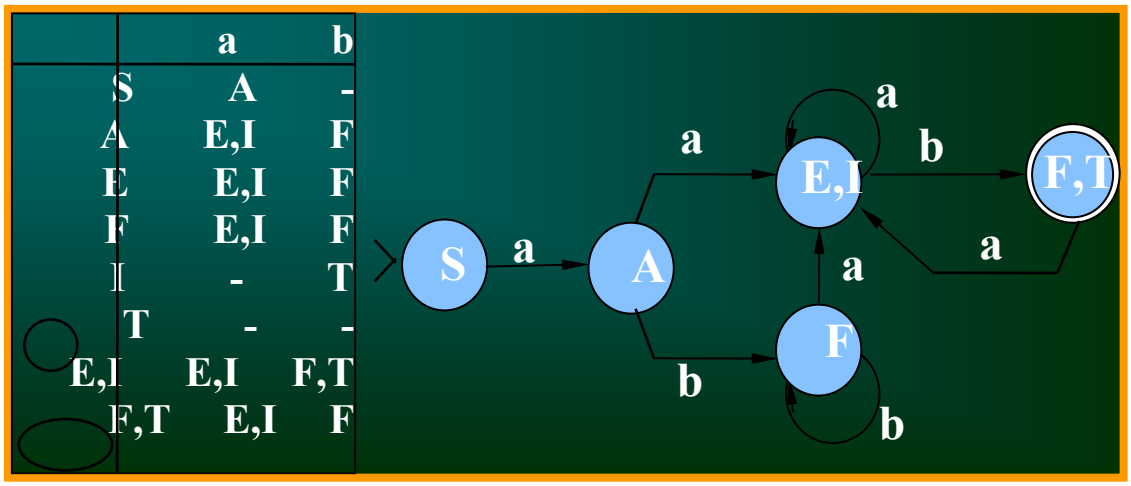
شكل 2.23- حذف گذرهاي تهي به حالات G ، C و D

پس از حذف گذرهاي تهي ماشين بصورت زير تبديل ميشود.



شكل 2.24- ماشين خودكارپس از حذف گذرهاي تهي

ماشين خودكار فوق در شكل 2.25 با استفاده از نمايش ماتريسي به فرم قطعي تبديل شده است.



شكل 2.25- ماشين خودكار قطعي براي عبارت a(a|b)*ab

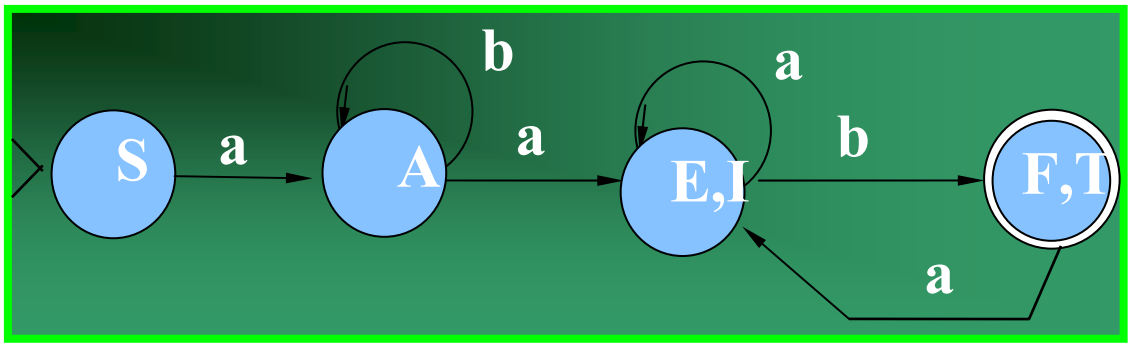
براي بهينه سازي ماشين خودكار فوق حالات مربوطه را به دو دسته پذيرشی و غير پذيرشی بصورت زير ميتوان تقسيم بندي نمود:

الف) 1- {FT} 2- {S, A, EI, F}

ب) 1- {FT} 2- {S} 3-{ A, EI, F}

ج) 1- {FT} 2- {S} 3-{ A, F} 4- {EI}

به اين ترتيب مشاهده ميكنيد كه حالات A و F با يكديگر معادل هستند و يكي را ميتوان با ديگري جايگزين نمود. به اين ترتيب ماشين خودكار بهينه براي عبارت a(a|b)*ab بصورت زير خواهد بود.



۲.۱۰ تمرين

تمرين 1- فرم كلي اعدادي را مشخص كنيد كه يا صفرمي باشندويا اينكه اگرطولشان بيش از يك است حتما" با يك رقم غير صفرآغاز مي شوند.

تمرين 2- فرم كلي اعداد در مبناي هشت را مشخص كنيددر صورتي كه بدانيم اينگونه اعداد حتماً با يك رقم صفر آغاز و سپس حرف **o** ظاهرمي‌گردد و بعد از آن عدد مبناي هشت مشخص مي شود. براي نمونه 0O127 يك عد مبناي 8 است.

تمرين 3- فرم كلي اعداد در مبناي شانزده در زبان C با رقم صفر وسپس X آغاز مي شود. يك عبارت با قاعده براي بيان اعداد در مبناي شانزده ارايه نماييد . بطور مثال 0XAF25 يك عدد در مبناي 16 است.

تمرين 4- فرم كلي رشته ها را در زبان C را با يك عبارت با قاعده معين كنيد. توجه داشته باشيد كه در زبان **C** علامت \ (Backslash) در داخل يك رشته مفهوم خاص دارد.

تمرين 5- فرم كلي اعداد صحيحي را تعيين كنيد كه در آنها ارقام به ترتيب صعودي ظاهر ميشوند.

تمرين 6- فرم كلي كليه رشته اعداد مبناي دو را مشخص كنيد كه در آنها زير رشته 011 ظاهر نگردد.

تمرين 7 *- كليه رشته هاي اعداد مبناي دو را مشخص كنيد كه تعداد صفرها در آنها فرد و تعداد يك ها زوج باشد.

تمرين 8- ماشين خودكار قطعي براي شناسه ها ترسيم نمائيد

تمرين 9- ماشين خودكار قطعي بهينه براي عبارت (a | b | c)* a b b ايجاد نماإيد و سپس برنامه تحليلگر لغوي براي تشخيص اينگونه رشته ها بنويسيد. ابتدا عبارت را به فرم غير قطعي تبديل نمائيد سپس قطعي و بهينه نمائيد.

تمرين 10- يك ماشين خودكار قطعي براي كليه اعداد مبناي دوكه تعداد صفرهاي آنها فردو تعداد يك ها فرد است ايجاد كنيد. در ضمن لااقل دو عدد صفر و دو عدد يك بايد در اين اعداد بايد موجود باشد.

تمرين 11- ماشين خودكار قطعي براي اعداد مبناي 2 كه حتما" داراي يك رقم صفر و يك رقم يك هستند راايجاد نمائيد بقسمي كه اعداد با تعداد يك ها و صفرهاي زوج را در يك حالت پذيرشی مشخص نمايد . اعداد با تعداد يك هاي فرد و تعداد صفرهاي فرد را نيز در حالت پذيرش ديگر بايد ماشين مشخص كند. براي اعداد با تعداد يك ها فرد و صفرها زوج و همچنين صفرها فرد و يك ها زوج نيز بايد دو حالت پذيرش مجزا وجود داشته باشد. ابتدا براي هر يك از چهار حالت ذكر شده ماشينهاي خودكار را ترسيم نمائيد و سپس يك ماشين خودكار غير قطعي براي تركيب آنها ايجاد نمائيد. سپس اين ماشين خودكار را قطعي كنيد.

تمرين 12- يك ماشين خودكار قطعي براي گرامر يكي از زبانهاي C يا پاسكال ايجاد نمائيد. سپس با استفاده از روش ارائه شده در اين فصل يك تحليلگر لغوي براي آن ايجاد كنيد.

فصل سوم

زبان وگرامرزبان

3.1 - گرامر زبانها :

زبانهاي برنامه سازي نيز همانند زبانهاي محاوره اي مبتني بر گرامر و قوانين خاص نحوي خود هستند . براي بيان قوانين گرامري زبانها از قوانين خاصي بنام **Bacus Normal Form** استفاده مي شود. اين فرم كه دراصطلاح يك **Meta Language** ناميده ميشود ، اولين بار براي بيان قوانين گرامري زبان **Algoal** استفاده شد . براي نمونه ساختارگرامري جملات را مي توان بصورت زير بيان كرد:

1) Statement ⟶ IfSt | WhileSt | RepeatSt | ForSt | CompoundSt | AssignmentSt | callst | CaseSt
2) IfSt ⟶ IF cond THEN statement ELSE part
3) ElsePart ⟶ ELSE statement | ε
4) WhileSt ⟶ WHILE cond DO statement
5) ForSt ⟶ FOR id := expression TO expression DO statement
6) CompoundSt ⟶ BEGIN statements END.
7) statements ⟶ statement | statements ‘;’ statement
8) AssignmentSt ⟶ id := expression
9) CallSt ⟶ id ‘(‘ ActualParams ‘)’ | id
10) CaseSt ⟶ CASE expression OF CaseList End
11) CaseList ⟶ CaseLabel ‘:’ Statement | ε
12) CaseLabel ⟶ Constant | CaseLabel , Constant
10) expression ⟶ expression ‘+’ term | expression ‘-’ term | expression Or term | term
11) term ⟶ term ‘*‘ factor | term ‘/’ factor | term And factor | factor
12) factor ⟶ id | No | ‘(‘ expression ‘)’ | Not factor
13) cond ⟶ expression | expression relop expression
14) relop ⟶ < | <= | > | >= | <>
15)ActualParams ⟶ expression | ActualParams ‘,’ expression

براي بيان قواعد گرامر فوق از چهار نوع علامت استفاده شده است :
1- علامت " ⟶ " به مفهوم " هست " ميباشد. براي نمونه قاعده شماره (4) مشخص ميكند كه يك جمله While داراي قاعده گرامري بصورت ارائه شده است .

2- علامت "|" به مفهوم "يا" است . براي نمونه (3) مشخص ميكند كه قسمت ELSE يا else part يا بصورت Else statment است و يا ا صلاً وجود ندارد كه با علامت اپسيلون ε نشان ميدهند .

3- علامت اپسيلون "ε" به مفهوم تهي ميباشد .

4- علائم پرانتز "(" و ")" معمولاً بعنوان جداكننده استفاده ميشوند. درگرامر فوق چون پرانتز بخشي از گرامر زبان ميباشد لذا ، آنرا در داخل كوتيشن قرار داده اند .

گرامر فوق از تعدادي قاعده و يا در اصطلاح Productions يا Rules تشكيل شده است. در سمت راست هر قاعده يك ترم مياني قرار گرفته و سپس علامت هست يا "→" و سپس گسترش آن ترم مياني در سمت چپ فلش مشخص شده است . براي مثال به قاعده شماره (7) توجه كنيد. دراين قاعده Statment سرترم گرامر است. اصولاً سه نوع ترم در داخل گرامر زبانها مشاهده ميشود .

1- ترم مياني (Non terminal) :

ترم مياني به آن دسته از ترمها اطلاق ميشود كه بنا به گرامر زبان داراي تعريف و قاعده باشد و يا بعبارت ديگر در سمت چپ لااقل يك قاعده قرار بگيرد. به ترم مياني ، ترم واسطه نيز گفته مي شود چرا كه واسطه اي براي بيان و مشخص نمودن قوانين گرامري است. براي مثال در زبان فارسي جمله ً اسناديً يك ترم مياني و در واقع واسطه اي براي بيان دستورالعمل هاي زبان فارسي است. در گرامر فوق ترمهايي مثل Cond , Expression و If st كه در سمت چپ قواعد قرار گرفته اند، ترمهاي مياني هستند.

2- ترمهاي پاياني (Terminal seymbols) :

اين دسته از ترمها اصولاً در سمت چپ قواعد ظاهر نمي شوند و در واقع لغاتي هستند كه در داخل زبان ظاهر مي شوند . كار تشخيص آنها به عهده تحليلگر لغوي ميباشد. از اين جمله مي توان شناسه ها، اعداد، جملات تفسيري كامنت و لغات كليدي زبان را نام برد. براي مثال در گرامر فوق ترمهاي پاياني از قبيل ELSE و For با حروف بزرگ مشخص شده اند.

2- **سر ترم گرامر (Start Seymbol) :**

سر ترم يا ترم آغازين گرامرها، ترمي است مياني كه شروع كننده يك گرامراست و نهايتاً در تعريف آن تمامي ترمها به نحوي گنجانده شده اند. براي نمونه درگرامر فوق Statement سر ترم گرامر است و IfSt سرترم نيست زيرا ، در تعريف Statement ترمي مانند IfSt يا بطور غير مستقيم ترمي مانند Relop وجود دارد اما در تعريف Ifst همواره كليه جملات بايد با كلمه If آغاز شوند. اصولاً گرامر ها يامستقل از متن[1] هستند و يا وابسته به متن مي باشند .

گرامرفوق مستقل ازمتن است. به اين مفهوم كه براي يك ترم مياني مهم نيست در چه متني و يا دركنار كدام ترم مياني ظاهر شود. يك ترم مياني مثل IfSt هميشه داراي ساختاري ثابت بوده ، مستقل از متني كه در آن ظاهر شده ، مي باشد. امّا، درگرامرهاي وابسته به متن يا Context sensetive ساختار يك ترم مياني ممكن است وابسته به اين كه چه ترم هايي در اطراف آن قرارگرفته اند ، فرق كند. براي مثال جملات If يا While مستقلاً ساختار خود را دارند. امّا، اگر جمله If درکنار جمله While ظاهر شود جمله حاصل ساختار ديگري خواهد داشت. به همين دليل است كه در گرامرهاي وابسته به متن در سمت چپ قواعد، ممكن است بيش از يك ترم مياني ظاهر شود.

درگرامر هاي مستقل از متن هميشه يك ترم مياني در سمت چپ هر قاعده قرار ميگيرد. مسلماً براي هر ترم مياني يك گسترش مي بايست وجود داشته باشد. لذا ، درگرامر وابسته به متن اگر در سمت چپ قاعده ، دو ترم مياني وجود داشته باشد در سمت راست بايد تعداد ترمها دو يا بيشتر باشد. اگر تعداد ترمها در سمت چپ قاعده بتواند بيش از تعداد ترم ها در سمت راست باشد آن گرامر را ديگر وابسته به متن نمي نامند. اين نوع گرامر را *نوع صفر* مي گويند .

نوع ديگر *گرامر باقاعده* است. در اين نوع گرامرها قواعد يا بصورت خود بازگشتي چپ و يا بصورت خود بازگشتي راست ميتوانند ظاهر شوند. البته قواعد غيرخودبازگشتي هم مي تواند وجود داشته باشد. براي نمونه قاعده شماره (7) درگرامر فوق خود بازگشتي راست است.

3.2 - درختهاي تجزيه :

هدف از ايجاد درختهاي تجزيه تحليل نحوي جملات است . به عبارت ساده تر با ايجاد درخت تجزيه صحت جملات از لحاظ قوانين گرامري مورد آزمون قرار

[1] مستقل از متن يا Context Free به گرامري اطلاق مي شود كه ساختار نحوي هر ترم مياني مستقل از متن دربر گيرنده آن است.

ميگيرد . اين عمل با تجزيه جملات براساس عناصر آنها صورت ميپذيرد . براي نمونه گرامر زير را در نظر بگيريد :

$E \rightarrow E + T \mid E - T \mid E \text{ OR } T \mid T$
$E \rightarrow T * F \mid T / F \mid T \text{ AND } F \mid F$
$F \rightarrow id \mid NO \mid (E) \mid \text{NOT } F$

گرامر فوق مبين ساختار كلي عبارات است ، بنابراين هر يك از عبارات چهار عمل اصلي بايد بر اساس اين گرامر ايجاد شده باشند. براي نمونه عبارت زير را بايد بتوان بر اساس گرامر فوق ايجاد نمود.

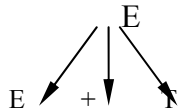
$a * (b - c) + d / (e - f)$

شايد روش تفكر يك معلم دستورالعمل زبان نيز به همين طريقي باشد كه توضيح داده خواهد شد. يعني اينكه جمله نوشته شده را تجزيه ميكند و مشخص مينمايد كه آيا براي مثال يك جمله انگليسي هست يا خير. معمولاً ذهن انسان به دو روش عمل مي نمايد. يك روش بالا به پائين است ، يعني اينكه با نگرش به جمله داده شده و اين نتيجه گيري كه جمله بايد يك عبارت باشد، عمل آغاز مي شود. عمل تجزيه از سر ترم گرامر يعني E آغاز مي شود. بناربراين در مرحله اول درخت تجزيه بصورت زير است :

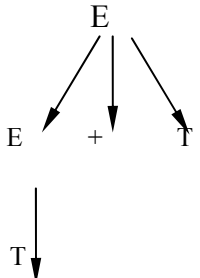
E

درخت تجزيه در مرحله 1

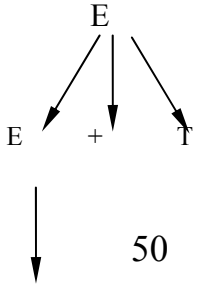
حالا با در نظر گرفتن اينكه عبارت داده شده حاصل جمع دو ترم ($a * (b - c)$ و $d / (e - f)$ است، سر ترم E بنابر قاعده $E \rightarrow E + T$ بصورت زير گسترش داده ميشود.

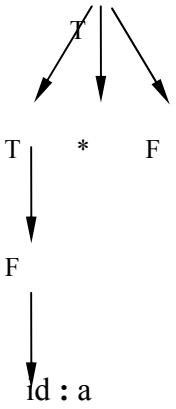


همانگونه كه در بالا توضيح داده شد عبارت داده شده حاصل جمع دو ترم است لذا ، بنا بر قاعده $E \rightarrow T$ ترم مياني E به T گسترش داده مي شود.

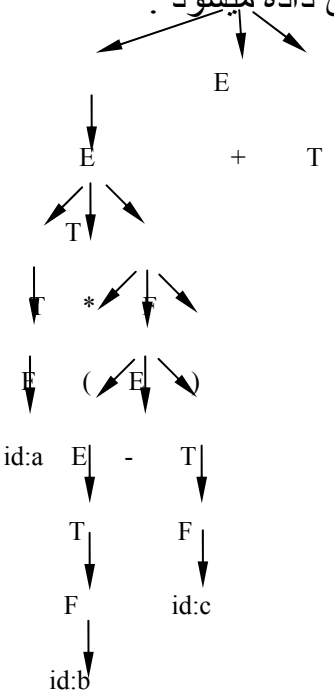


اكنون بايد بتوان از ترم مياني T عبارت $a * (b - c)$ را توليد كرد لذا ، درخت تجزيه فوق بصورت زير نوسعه داده ميشود.

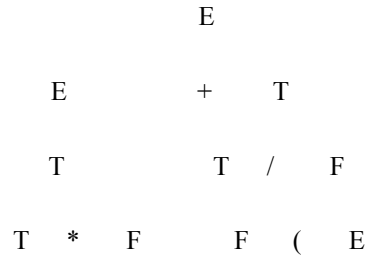


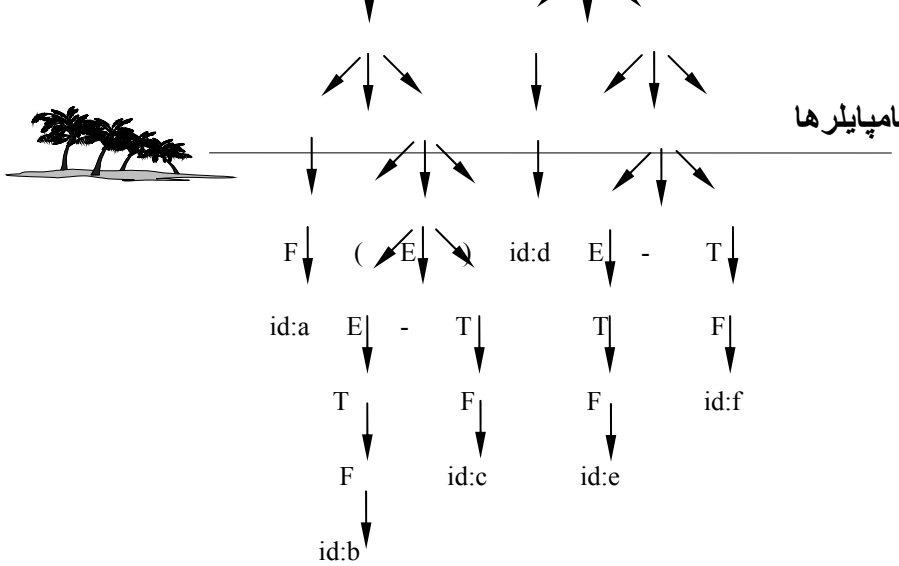


اکنون باید بتوان F را با (b - c) جایگزین کرد. بنابراین در خت تجزیه فوق بصورت زیر گسترش داده میشود :



اکنون با توجه به اینکه پس از علامت جمع باید از ترم میانی T نهایتاً عبارت d / (e - f) حاصل شود درخت تجزیه بصورت زیر توسعه داده میشود:





همانگونه كه در شكلهاي فوق مشاهده ميشود با استفاده از روش بالا به پائين و تجزيه يك عبارت بر طبق گرامر نهايتاً درخت تجزيه كه در رأس آن سر ترم گرامر و در برگها ترمهاي پاياني قرار دارند ايجاد شده است .
چنانچه عبارت از لحاظ گرامري غلط مي بود، اين امكان وجود نداشت كه بتوان بر اساس گرامر درخت فوق را ايجاد كرد. مراحل تجزيه با ايجاد مرحله به مرحله درخت نمايش داده شد زيرا، با توجه به درخت تجزيه نهايي نمي توان مراحل تجزيه را مشخص نمود. در طي مراحل ايجاد درخت تجزيه ، عمل تجزيه ترمهاي مياني از چپ به راست انجام شده ، همواره سمت چپ ترين عنصر يا گره در داخل درخت گسترش داده شد تا نهايتاً بتوان به يك ترم پاياني بعدي در داخل جمله داده شده رسيد. به عبارت ديگر درخت تجزيه مشخص نميكند كه در طي چه مراحلي جمله داده شده از سر ترم گرامر مشتق يا نتيجه گيري شده است . مراحل تجزيه سر ترم تا رسيدن به ترمهاي پاياني درون جمله داده شده را در اصطلاح مراحل اشتقاق يا Deriivation مي گويند. مراحل اشتقاق سر ترم E تا رسيدن به جمله فوق به صورت زير است :

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow T*F+T \Rightarrow F*F+T \Rightarrow a*F+T \Rightarrow a*(E)+T \Rightarrow$$
$$a*(E-T)+T \Rightarrow a*(T-T)+T \Rightarrow a*(F-T)+T \Rightarrow a*(b-T)+T \Rightarrow$$
$$a*(b-F)+T \Rightarrow a*(b-c)+T \Rightarrow a*(b-c)+T/F \Rightarrow a*(b-c)+F/F \Rightarrow$$
$$a*(b-c)+d/F \Rightarrow a*(b-c)+d/(E) \Rightarrow a*(b-c)+d/(E-T) \Rightarrow$$
$$a*(b-c)+d/(T-T) \Rightarrow a*(b-c)+d/(F-T) \Rightarrow a*(b-c)+d/(e-T) \Rightarrow$$
$$a*(b-c)+d/(e-F) \Rightarrow \mathbf{a*(b-c)+d/(e-f)}$$

همانگونه كه مشاهده نموديد، عمل اشتقاق از سر ترم آغاز و نهايتا به جمله داده شده خاتمه مي يابد. در اين ميان فرمهاي جمله أي كه حاوي ترمهاي مياني و پاياني و يا فقط ترمهاي مياني هستند، ايجاد مي‌گردد. فرمهاي جمله اي را **Sentential form** نيز ميگويند .

3.4 تجزيه پائين به بالا

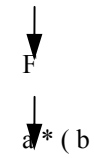
در روش تجزيه پائين به بالا بر خلاف روش بالا به پائين ، عمل تجزيه از پائين و از ترمهاي درون جمله آغاز و به سر ترم گرامر خاتمه مي يابد. لذا ، اين روش پائين به بالا است. براي نمونه عبارت زير را در نظر ميگيريم :

a * (b - c) + d

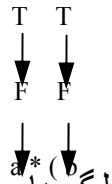
هدف تجزيه اين عبارت به روش پائين به بالا است . همانگونه كه توضيح داده شد در روش پائين به بالا عمل تجزيه از چپ به راست با جايگزينی ترمهاي پاياني با مياني بر اساس سمت چپ قواعد انجام مي شود و آنقدر عمل جايگزينی ادامه مي يابد تا درصورت صحت ، بتوان به سرترم داده شده رسيد .

ابتدا سمت چپ ترين لغت از عبارت داده شده يعني a توسط تحليلگرلغوي خوانده مي شود. بر سر ورودي علامت * ظاهر مي شود. طبق قاعده $T \rightarrow T * F$ ، قبل از عملگر * فقط يك T مي تواند ظاهر شود. لذا ، با استفاده از قاعده $F \rightarrow id$ ترم پاياني a كه يك شناسه يا id است را با F و سپس طبق قاعده $T \rightarrow F$ ترم مياني F را با T مي توان جايگزين نمود. اكنون ترم بعدي يعني * از سر ورودي خوانده مي شود. طبق قاعده $T \rightarrow T * F$ پس از * ترم مياني F مي تواند ظاهر شود. لذا ، ترم بعدي بايستي F باشد .

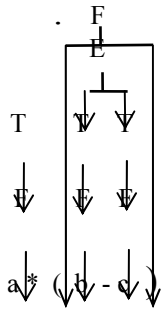
اكنون ورودي ' **(b - c) + d** '، است. بر سر ورودي لغت (وجود دارد. پس ازخواندن (و b درخت بصورت زير خواهد بود :



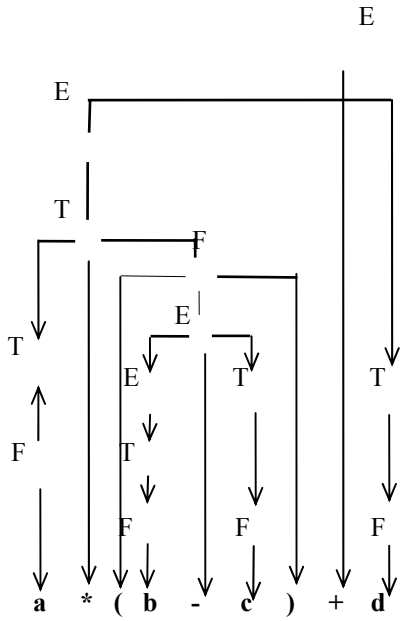
حال بر سر ورودي لغت منهاوجود دارد. طبق قاعده $E \rightarrow E - T$ قبل از عملگر - بايد يك عبارت E وجود داشته باشد. لذا، ترم پايانی b كه يك شناسه يا id است بنابر قاعده $F \rightarrow id$ با ترم مياني F و سپس طبق قواعد $T \rightarrow F$ و $E \rightarrow T$ نهايتاً با ترم مياني E جايگزين مي شود. بنابراين ، تا اين مرحله درخت تجزيه پائين به بالا بصورت زير خواهد بود.



اكنون ورودي ' d + (c - ' ، است. عملگرهاي - و سپس c از ورودي خوانده مي شوند. طبق قاعده $E \rightarrow E - T$ بايد پس از - در ورودي يك ترم ظاهر شود. لذا ، c با F و با T و نهايتاً E - T با E جايگزين خواهند شد . حال (از ورودي خوانده مي شود و به اين ترتيب درداخل درخت تجزيه (E) ظاهر مي شود و طبق قاعده $F \rightarrow (E)$ ميتوان آنرا با F جايگزين كرد.



اكنون ورودي ’d +‘ است. برسرورودي علامت + وجود دارد. قبل از + طبق قاعده$E \to E + T$ بايد يك E وجود داشته باشد. بنابراين T * F را كه اكنون در رأس درخت تجزيه وجود دارد طبق قاعده $T \to T * F$ با T و سپس طبق قاعده $E \to T$ ترم T را با E مي توان جايگزين نمود. نهايتاً درخت تجزيه به صورت زير تبديل مي شود :



همانگونه كه مشاهده نموديد عمل تجزيه از ترمهاي پاياني آغاز و به سر ترم گرامر خاتمه يافت . اين گونه تجزيه را در اصطلاح تجزيه پائين به بالا يا Bottom Up Parsing گويند .

طبق تعريف ، مراحل اشتقاق نمايانگر مراحل رسيدن از سر ترم گرامر به ترمهاي پاياني درون جمله مورد نظر است در تجزيه پائين به بالا ، مراحل اشتقاق درست بر خلاف مراحل تجزيه است چرا كه در اينجا نهايتا به سر ترم گرامر مي رسند در صورتي كه مراحل اشتقاق از سر ترم گرامر آغاز ميشود. به عبارت ديگر مراحل اشتقاق نشان مي دهد كه در طي چه مراحلي جمله داده شده از سر ترم گرامر مشتق شده است. مراحل مشتق كردن عبارت ’a * (b - c) + d‘ از سرترم E

بصورت زير است. بايد توجه داشته باشيد كه مراحل اشتقاق درست بر خلاف مراحل تجزيه است . بنابراين اگر از آخرين مرحله ايجاد درخت تجزيه به مراحل قبلي برگشت شود مراحل اشتقاق بصورت زير معين مي شود :

$$E \Rightarrow E+T \Rightarrow E+F \Rightarrow E+d \Rightarrow T+d \Rightarrow T*F+D \Rightarrow T*(E)+d \Rightarrow T*(E-T)+d \Rightarrow T*(E-F)+D \Rightarrow T*(E-c)+d \Rightarrow T*(T-c)+d \Rightarrow T*(F-c)+d \Rightarrow T*(b-c)+d \Rightarrow F*(b-c)+d \Rightarrow a*(b-c)+d$$

همانگونه كه مشاهده مي شود در طي مراحل اشتقاق بر خلاف اشتقاق چپ كه قبلاً توضيح داده شد، همواره سمت راست ترين ترمها تا رسيدن به ترم پايانی درون جمله داده شده جايگزين مي گردند لذا اين گونه اشتقاق را *اشتقاق راست* گويند . اولين ترم پايانی كه در طي اين مراحل اشتقاق در فرمهاي جمله اي ظاهر شد ترم پايانی d در فرم جمله اي $T+d$ بود و سپس از راست به چپ ترم پايانی بعدي يعني c در فرم جمله أي $T*(E-c)+d$ ظاهر گرديد لذا همانگونه كه مشاهده مي كنيد جايگزينی از سمت راست به چپ انجام شده است .

3.5 گرامرهاي مبهم

چنانچه بتوان براي جمله داده شده بر اساس گرامر زبان ، بيش از يك درخت تجزيه ايجاد نمود، آن گرامر را مبهم مي نامند. در حالت كلي با نگرش به گرامرها نمي توان ابهام را تشخيص داد. اما بايد ديد كه چرا اگر بتوان بيش از يك درخت تجزيه براي جمله داده شده ايجاد نمود، گرامر ابهام دارد . اين بايد يك مزيت باشد، چرا آنرا ابهام ميخوانند ؟ به عنوان يك مثال ساده به گرامر زبانهاي C يا پاسكال براي جملات شرطي if توجه نماييد :

statement → ifSt | whileSt | compoundSt | assignmentSt | callSt |
ifSt → IF cond THEN statement | IF cond THEN statement ELSE statement
cond → exp | exp Relop exp

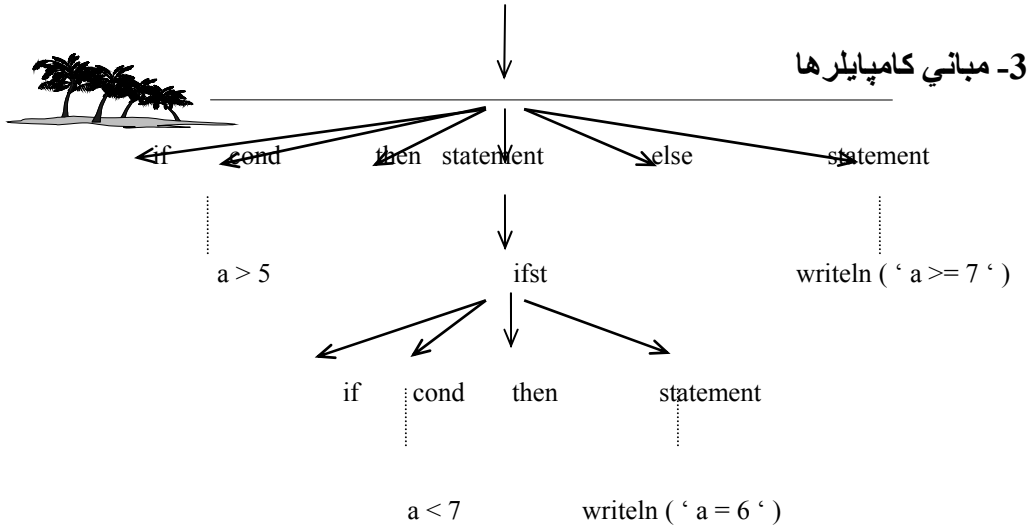
براي نمونه به جمله زير توجه نماييد :

IF a > 5 THEN IF a < 7 THEN writeln (‘ a = 6 ‘) ELSE writeln (‘ a >= 7 ‘)

بنا بر گرامر فوق درخت تجزيه بصورت زير مي تواند باشد :

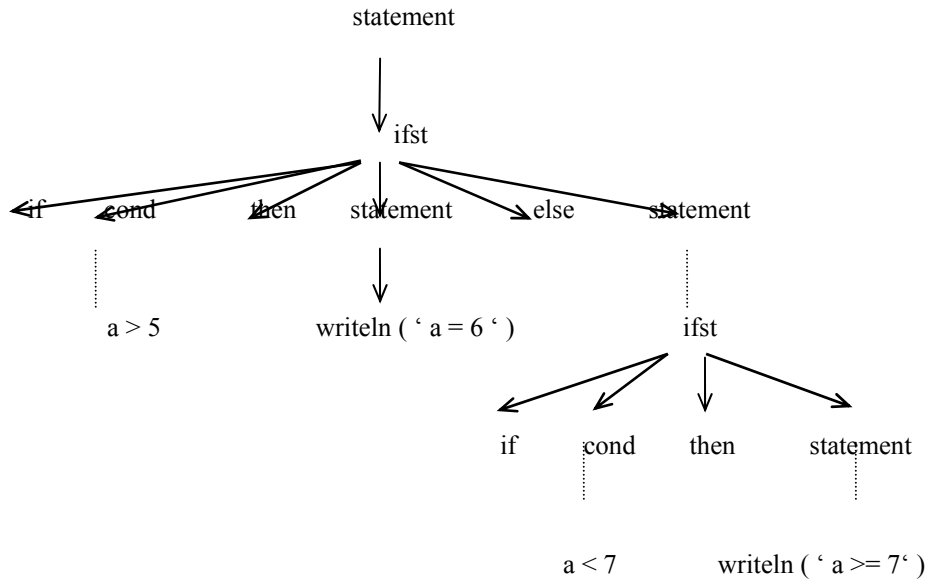
statement

ifst



شكل 3.1- درخت تجزيه براي جمله if

درخت تجزيه را بصورت زير نيز مي توان توليد كرد :



شكل 3.2- درخت تجزيه براي جمله if

بنابر گرامرفوق دو درخت تجزيه با دو مفهوم متفاوت ايجاد شده است . در درخت تجزيه شكل 3و5 ، else وابسته به if خارجي ميباشدودر شكل 3و6 ، else وابسته به if داخلي است . براي رفع اين مشكل در زبانهاي C و پاسكال ، براي تكميل گرامر بطور ضمني و نه بر طبق گرامر ، تعيين كرده اند كه else به نزديكترين if وابسته است . بنابراين درخت تجزيه شكل 3و6 توسط كامپايلر پاسكال ايجاد ميشود. در زبان فورترن 77 با افزايش Endif اين مشكل حل شده و گرامر بصورت زير تبديل گرديده است :

ifst → IF cond THEN statement ENDIF
| IF cond THEN statement ELSE statement ENDIF

بنابراين اگرقراراست كه درجمله داده شده else وابسته به if خارجي باشد، به صورت زير عمل ميشود :

IF a > 5 THEN IF a < 7 THEN writeln (‘ a = 6 ‘) ENDIF ELSE writeln (‘ a >= 7 ‘) ENDIF

بالعكس اگر else وابسته به if داخلي باشد ، جمله بصورت زير نوشته مي شود :

IF a > 5 THEN IF a < 7 THEN writeln (‘ a = 6 ‘) ELSE writeln (‘ a >= 7 ‘) ENDIF ENDIF

اصولاً درحالت كلي ابهام در گرامر ها را نمي توا ن نشان داد و اثبات نمود اما نكته قابل توجه اين است كه اگر گرامري بصورت خود بازگشتي چپ وخود بازگشتي راست براي يك ترم مياني قواعدي داشته باشد آن گرامر مبهم است.

1) $A \rightarrow A\ \alpha \mid \beta\ A$

همچنين اگر در يك گرامر نهايتاً بتوان طي مراحل استنتاج يا اشتقاق از يك ترم مياني به خود آن ترم مياني رسيد، گرامر مبهم ميباشد.

2) $A \Rightarrow \alpha \Rightarrow \ldots \Rightarrow A$

اگر در گرامري قواعد بصورت خود بازگشتي راست يا چپ براي يك ترم مياني وجود داشته باشد و علاوه بر آن قاعده ديگر براي اين ترم مياني بصورتي باشد كه حالت خود بازگشتي در وسطح دو ترم در سمت راست آن قاعده براي آن ترم مياني وجود داشته باشد و در اصطلاح قاعده بصورت Self Embeding ياخودگردان باشد ، بازهم گرامر مبهم خواهد بود.

3) $A \rightarrow A\alpha \mid \beta A\ \delta$

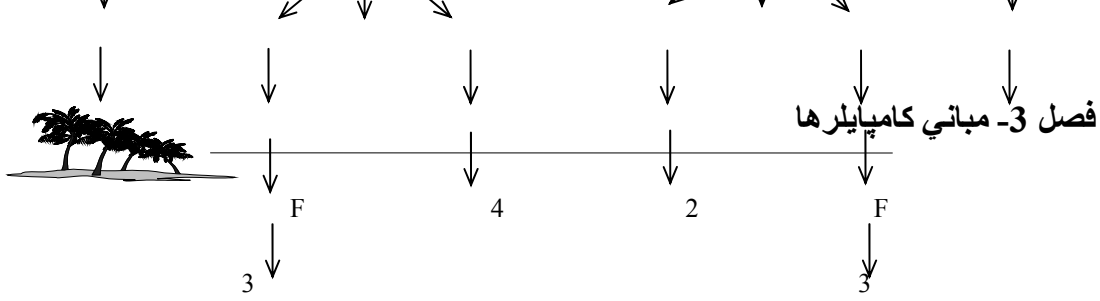
براي نمونه به گرامرمبهم عبارات توجه كنيد :

$E \rightarrow E + T \mid T - E \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid id \mid NO$

مي خواهيم براي جمله 2 - 3 + 4 درخت تجزيه ترسيم نماييم :



الف- درخت با حاصل: $5 = 4 + (2 - 3)$ **ب-** درخت با حاصل: $-5 = (4 + 3) - 2$

شكل 3.3- دو درخت تجزيه متفاوت براي يك عبارت

همانگونه كه مشاهده ميكنيد ايجاد دو نتيجه متفاوت 5 و 5- براي عبارت فوق بخاطر مبهم بودن گرامر و در واقع وجود قاعده بصورت خود بازگشتي چپ و خود بازگشتي راست براي ترم مياني E است .

3.6 تمرين

تمرين 1- مراحل استنتاج را براي عبارت **(a * b - c) / (d - e)** در روشهاي تجزيه بالا به پائين و پائين به بالا مشخص نماييد .

تمرين 2- گرامر ساختار ليست را در نظر بگيريد:

$S \rightarrow$ '(' L ')' | a
$L \rightarrow L$ ',' S | S

درخت تجزيه را براي جملات زير ترسيم نمائيد :

الف - (a , a)

ب - (a, (a, a))

ج - (a, ((a, a), (a, a))

تمرين 3- نشان دهيد كه گرامر زير مبهم است :

$S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \varepsilon$

تمرين 4- گرامر زير را در نظر بگيريد :

bexpr $\rightarrow$ bexpr *or* bterm | bterm
bterm $\rightarrow$ bterm *and* bfactor | bfactor
bfactor $\rightarrow$ *not* bfactor | (bexpr) | *true* | *false*

الف - درخت تجزيه براي عبارت not(true or false) ايجاد كنيد.
ب - آيا اين گرامر مبهم است.

تمرين 5- چرا اگر در يك گرامر براي يك ترم مياني قواعد به دو صورت خودبازگشتي چپ و خودبازگشتي راست وجود داشته باشند آن گرامر مبهم است.

فصل چهارم

تجزيه بالا به پائين

٤.١ تحليلگر ذهن

عمل تجزيه بالا به پائين بر مبناي روش تفكر مغز آدمي براي تحليل نحوي جملات استوار است. فردي فارسي زبان در مقابل شما قرار ميگيرد . ميخواهد با شما صحبت كند. به دهان وي مينگريد. پيش بيني مي كنيد كه جمله فارسي ميخواهد بيان كند. جمله فارسي سرترم جملات در دستورالعمل زبان فارسي است زيرا ، هر گفته اي در زبان فارسي بايد جمله فارسي باشد. بنابراين مغز پيش بيني ميكند كه جمله خارج شده از دهان سرترم گرامر جملات در زبان فارسي بايد باشد.

حالا فرض كنيد كه فرد كلمه ،اگر، از دهانش خارج شود. بلافاصله تحليلگر نحوي مغز به سراغ مجموعه لغات يا در اصطلاح ترمهاي پايانی اي ميرود كه مي توانند جملات فارسي را آغاز كنند. كلمه " گر" ميتواند آغاز كننده يك جمله فارسي باشد. در اصطلاح مجموعه ترم هاي پايانی كه ميتوانند آغاز كننده يك ترم باشند را مجموعه سرآغاز يا مجموعه First براي آن ترم گويند.

پس از اطمينان از اينكه لغت ،اگر، در مجموعه سرآغاز يا First جمله فارسي است بايد مشخص نمود كه كدام گسترش از گسترش‌هاي متفاوت جمله فارسي با كلمه ،گر، آغاز ميشود. پاسخ جملات سوألي است. كلمه اگر متعلق به مجموعه سرآغاز يا First جملات سوألي است. حالا طبق گرامر جملات سوألي پس از كلمه ،اگر، انتظار شنيدن يك شرط ميرود. باز هم بر طبق گذشته تحليلگر نحوي ذهن كلمه بعدي را گرفته در مجموعه سرآغاز شرط آنرا جستجو ميكند. و به اين ترتيب مراحل ادامه مي يابد.

٤.٢ ايجاد الگوريتم تحليل نحوی بر مبنای عملكرد ذهن

بطور خلاصه در بخش قبل ملاحظه نموديد كه براي انجام عمل تحليل نحوي مغز به صورت زير عمل مي نمايد :

1. انتظار سرترم گرامر زبان را در آغاز دارد. سرترم گرامر زبان فارسي جمله فارسي است.

2.لغت اولي دريافت ميشود . در بخش قبل در حاليكه انتظار جمله فارسي را تحليلگر ذهن مي داشت لغت "گر" دريافت شد.
3.در داخل مجموعه First براي ترم مورد انتظار بدنبال لغت دريافت شده ميگردد.
4.در صورت يافتن لغت ، در داخل مجموعه First براي گسترش‌هاي متفاوت ترم مياني مورد انتظار مي گردد تا مشخص كند كدام گسترش با لغت دريافت شده آغاز ميشود (در مثال فوق جمله شرطي با لغت گر، آغاز مي شد).
5. حالا با مشاهده ترم پايانی دريافت شده در گرامر ، طبق گرامر ترم مورد پيش بيني را مشخص ميكند. در مثال فوق با يافتن كلمه گر، طبق گرامر انتظار مشاهده يك شرط مي رفت. لذا ، لغت بعدي دريافت شده و در صورت عدم خاتمه جمله از مرحله 2 عمليات تكرار ميشود.

براي نمونه به گرامر زير توجه كنيد :

Statement → IfSt | WhileSt | ForSt | CaseSt | CompoundSt | AssignmentSt | CallSt

IfST → IF Expression THEN Statement
ElsePart

ElsePart → ELSE Statement |
∈

WhileSt → WHILE Expression DO
Statement

ForSt → FOR Variable := Expression TO Expression DO Statement

CaseSt → CASE Expression OF CaseParts
END

CaseParts → CaseLabels : Statement
OtherParts

OtherParts → ; CaseParts |
∈

CaseLabels → Constant
Constants

Constants → , CaseLabels |
∈

Constant → String | Identifier |
Number

CompoundSt → BEGIN Statements
END

Statements → Statement
OtherSts

OtherSts → ; Statements | ∈

AssignmentSt → Id :=
Expression

CallSt → Id '(' ActualParams

')'

ActualParams → Expression

OtherParams

OtherParams → , OtherParams | ∈

Expression → Id |

No

براي تجزيه جمله

If Then C := 1

A

بر اساس عملکرد ذهن بصورت زیر عمل میشود.

- بر اساس گرامر در ابتدا انتظار مشاهده سرترم گرامر يعني Statement ميرود.
- اولين لغت يعني If از تحليلگر لغوي دريافت ميشود.
- تحليلگر نحوي به جستجوي لغت If داخل مجموعه لغات آغاز كنده ترم مورد انتظار يعني

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]

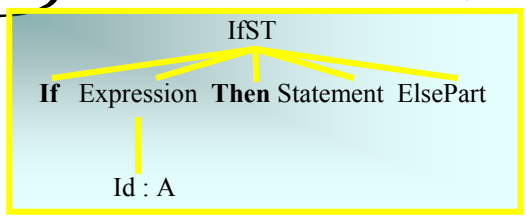
- با مشاهده نوع لغت If در مجموعه First(statement) تحليلگر لغوي بدرون مجموعه سرآغاز براي گسترش‌هاي متفاوت Statement مينگرد و لغت if را در مجموعه First(IfSt) پيدا ميكند. بنابراين تحليلگرتشخيص ميدهد كه جمله مورد نظر يك جمله شرطي If است. درخت زير بر اساس گسترش موجود براي جمله IfSt بر طبق گرامر ، ايجاد ميشود :

- حالا پس از مشاهده If ، بنابر گرامر جمله شرطي IfSt در ورودي انتظار مشاهده Expressiom ميرود. از تحليلگر لغوي لغت بعدي يعني A دريافت ميشود. درون مجموعه سرآغاز عبارات يعني :

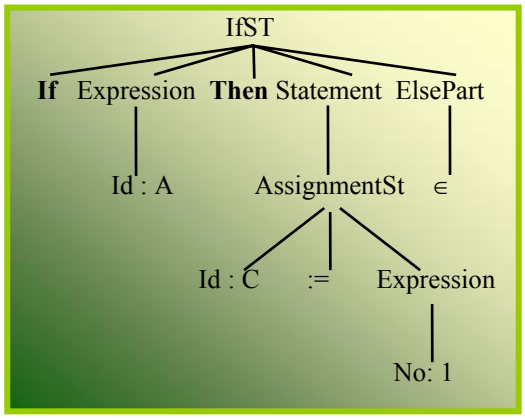
First(Expression) = [S_Id]

نوع لغت A يعني S_Id وجود دارد. بنابراين تحليلگر نحوي به كار خود ادامه ميدهد.

- حالا طبق گرامر انتظار ديدن Then در ورودي ميرود. تحليلگر لغوي لغت بعد از A را از ورودي ميخواند . اين لغت همانگونه كه طبق گرامر انتظار مي رفت لغت Then ميباشد. بنابراين تا اين لحظه درخت تجزيه زير توسط تحليلگر نحوي بنا شده است:



- حالا تحليلگر پيش بيني مشاهده يك Statement را بر اساس گرامر مينمايد. از تحليلگر لعوي لغت بعدي را تحليلگر نحوي دريافت ميكند. لغت C از نوع S_Id را تحليلگر نحوي دريافت ميكند. اين لغت در مجموعه First(Statement) وجود دارد اما ، نكته اينجا است كه هر دو گسترش CallSt و AssignmentSt با Id آغاز ميشوند. لذا ، تحليلگر نحوي نمي تواند بر اساس ترم پيش بيني Id تصميم بگيرد كه كدام گسترش بايد انتخاب شود. بنابراين ترم بعدي را دريافت مي كند ترم بعدي در مثال فوق := است. حالا ميتوان تصميم گرفت كه گسترش AssignmentSt بايد انتخاب شود. نهايتا" ، درخت تجزيه بصورت زير خواهد بود.



٤.٣ نتيجه تحليل عملكرد ذهن

از مطالب ارائه شده در بخش قبلي مي توان سه نتيجه بشرح زير گرفت:
اولا" در هر مرحله تحليلگر پيش بيني ميكند كه ترم بعدي چه بايد باشد. براي نمونه در ابتدا پيش بيني سر ترم گرامر را تحليلگر نحوي مي نمود. به اين نوع تحليلگرها در اصطلاح تجزيه گرهاي پيش بيني كننده يا Predictive Parsers گويند.
ثانيا" عمل خواندن لغات از چپ به راست صورت ميگيرد. درمثال فوق ابتدا سمت چپ ترين لغت يعني If از ورودي خوانده شد و عمل خواندن از چپ به راست ادامه پيدا كرد. لغت خوانده شده از ورودي را در اصطلاح ترم پيش بيني يا Look Ahead گويند زيرا ، بر اساس اين لغت است كه تحليلگر پيش بيني ميكند ترم بعدي چه بايد باشد. براي نمونه بر اساس ترم پيش بيني If تحليلگر نحوي تصميم گرفت كه گسترش IfSt را براي Statement بايد انتخاب نمايد. ثس پيش بيني مشاهده جمله IfSt در ورودي شد.

ثالثا" با در دست داشتن يك يا بيشتر از ترم هاي پيش بيني تحليلگر مي تواند تصميم گيري كند كه كدام گسترش از گسترش‌هاي متفاوت ترم مورد انتظار را بايد انتخاب نمايد. براي نمونه در بالا با در دست داشتن يك ترم پيش بيني If تحليلگر تصميم گرفت كه گسترش IfSt را براي ترم مورد پيش بيني يعني Statement بايد انتخاب كند اما، در هنگاميكه در ورودي ترم پيش بيني از نوع Id ظاهر شد چون دو گسترش متفاوت Statement يعني AssignmentSt و CallSt هر دو با ترم Id آغاز ميشدند ، تحليلگر نمي توانست تصميم بگيرد كه كداميك را انتخاب كند. در اين حالت ، با در دست داشتن دو ترم پيش بيني Id و := تحليلگر توانست تصميم بگيرد كه كدام گسترش بايد انتخاب شود.

اصولا" عمل تجزيه بالا به پائين از سر ترم گرامر آغاز و به ترمهاي پايانی درون جمله يا برنامه مورد كامپايل خاتمه مي يابد. براي انجام عمل تجزيه بالا به پائين فرم كلي جملات يا بعبارت ديگر گرامر زبان را آنقدر محدود ميكنند كه ، با در دست داشتن k ترم‌پيش بيني از متن برنامه مورد كامپايل بتوان عمل تحليل تحوي را بدرستي انجام داد.

تجزيه گرهاي پيش بيني كننده يا Predictive Parsers با در دست داشتن يك يا چند ترم پايانی بعدي در ورودي ، قادر به انجام عمل تحليل نحوي هستند . گرامرهاي مورد استفاده براي اين روش تجزيه را اصطلاحاً LL (k) يا Left Lookahead گويند . منظور اينست كه مي توان با استفاده از قواعد گرامر و با در دست داشتن k ترم پايانی بعدي در ورودي يا در اصطلاح k ترم پيش بيني ، عمل تجزيه بالا به پائين را انجام داد.

٤.٤ گرامرهای LL (1)

تجزيه گرهاي پيش بيني كننده براي تجزيه بالا به پائين با در دست داشتن يك يا چند ترم بعدي در ورودي بايد قادر به تشخيص اين باشند كه :

اولاً : جمله مورد نظر تا اين مرحله از لحاظ گرامري صحيح است .
ثانياً : چه ترمهاي مياني مورد انتظار هستند .
ثالثاً : چه ترمهاي پايانی در سر ورودي مي توانند ظاهر شوند .

چنانچه با در دست داشتن حداكثر K ترم بعدي از ورودي يا بعبارت ديگر با در دست داشتن حداكثر K ترم پيش بيني بتوان عمل تجزيه قابل پيش بيني را بر روي يك گرامر به انجام رساند آن گرامر را LL(K) گويند.

چنانچه با در دست داشتن يك ترم پايانی بعدي از چپ به راست در ورودي بتوان تشخيص داد كه از بين گسترش هاي متفاوت براي يك ترم مياني كداميك بايد انتخاب شوند ، گرامر را LL (1) گويند .

براي نمونه گرامر زير را در نظر بگيريد :

$S \rightarrow I \mid W \mid A \mid P \mid C$

$I \rightarrow$ if B do S E

$B \rightarrow$ id D

$D \rightarrow \varepsilon \mid R$ id

$R \rightarrow < \mid > \mid =$

$W \rightarrow$ while B do S

$A \rightarrow$ id := No

$P \rightarrow$ id '(' M ')'

$M \rightarrow \varepsilon \mid$ id

$C \rightarrow$ '{' T '}'

$T \rightarrow S\ G$

$G \rightarrow ; T \mid \varepsilon$

$E \rightarrow$ else S $\mid \varepsilon$

اين گرامر LL (1) نيست زيرا ، براي نمونه با ديدن id يا شناسه نميتوان مشخص نمود كه از بين قواعد مختلف براي گسترش S كداميك بايد انتخاب شود. اصولاً جهت تجزيه بالا به پائين بايد پس از خواندن يك ترم پايانی از ورودي مشخص نمود كه آيا ترم پايانی خوانده شده متعلق به مجموعه First براي ترم پايانی يا مياني مورد انتظار است يا خير ؟ . سپس بايد مشخص نمود كه كدام گسترش از گسترش هاي متفاوت ترم مياني مورد انتظار با ترم پايانی خوانده شده آغاز مي شوند. به عبارت ديگر بايد مشخص نمود كه ترم پايانی خوانده شده متعلق به مجموعه First براي كداميك از گسترش‌هاي مختلف ترم مياني مورد نظر است .

بنا بر اين اگر قرار باشد با داشتن يك ترم پيش بيني در هر مرحله از تجزيه بتوان مشخص نمود كه كدام گسترش براي ترم مياني مورد انتظار A با گسترش‌هاي

$A \rightarrow A1 \mid A2 \mid .... \mid An$

بايد انتخاب شود ، آنگاه بايد اشتراك مجموعه First براي هر دو گسترش متفاوت از A تهي باشد يا بعبارت ديگر بايد رابطه :

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

برقرار باشد. در غير اينصورت فرض كنيد كه براي نمونه

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

آنگاه با مشاهده ترم پایانی a در ورودي تحليلگر نمي تواند بلافاصله تصميم بگيرد كه آيا گسترش A به Ai بايد انتخاب شود. يا گسترش A به Aj.

پس بطور خلاصه ميتوان گفت :

يك گرامر LL(1) است اگر اشتراك مجموعه first هر دو گسترش متفاوت براي هر ترم مياني متعلق به آن گرامر تهي باشد .

براي نمونه درخت تجزيه را براي جمله :

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

با روش بالا به پائين ميتوان بصورت زير ايجاد نمود :

1- تحليلگر لغوي فراخواني ميشود .

2- لغت '{' توسط تحليلگر لغوي به تحليلگر نحوي برگردانده ميشود .

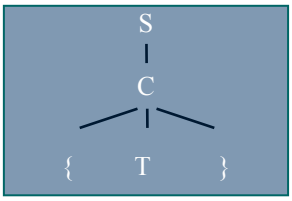
3- تحليلگر نحوي كه در انتظار مشاهده S در ورودي است ، ابتدا ميبايست مطمئن شود كه ترم خوانده شده يعني '{' آيا متعلق به مجموعه first (S) است.

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

سپس بايد مشخص شود كه '{' به مجموعه First براي كدام گسترش از گسترش‌هاي متفاوت S تعلق دارد . چون :

'{' ε first (C) = { '{' }

پس درخت تجزيه زير ايجاد ميشود :

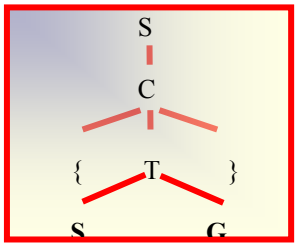


4- با مشاهده { در ورودي دو مرتبه مراحل 1 تا 4 بشرح زير تكرار مي شود .

- لغت بعدي a ميباشد كه از نوع id است .

id ∈ first (T) = first (S) = { if , while , id , '{' }

بنابر اين گسترش S به T انتخاب شده ، درخت تجزيه بصورت زير خواهد بود.



اما در اين مرحله مشاهده ميشود كه دو گسترش مختلف $S \rightarrow A$ و $S \rightarrow P$ هر دو با يك ترم id آغاز ميشوند بنابراين در اين مرحله به علت LL(1) نبودن گرامر

نمیتوان با در دست داشتن ترم پیش بیني a تصمیم قطعي در مورد گسترش S گرفت.

همانگونه که در بالا توضیح داده شد ، براي انجام تجزیه بالا به پائین نیاز به در دست داشتن مجموعه First براي ترمهاي گرامر است. روش محاسبه مجموعه سرآغاز یا مجموعه First در بخش بعدي توضیح داده خواهد شد .

٤.٥ مجموعه های سرآغاز و پیرو

مجموعه سرآغاز براي يك ترم شامل مجموعه ترمهاي پایانی است که گسترش‌هاي متفاوت و جملات مشتق از آن ترم را مي توانند آغازکنند. در حالت کلي براي ترم A مجموعه سرآغاز بصورت زیر محاسبه میشود:

- چنانچه براي A گسترش‌ي بصورت $A \rightarrow a\,\alpha$ وجود داشته باشد ، آنگاه :
  $First(A) = \{ a \}$

- چنانچه براي A گسترشی بصورت $A \rightarrow B\,\alpha$ وجود داشته باشد ، آنگاه :
  $First(A) = First(B)$

- چنانچه براي A گسترشی بصورت $A \rightarrow B\,\alpha$ وجود داشته باشد و $B \rightarrow \delta \mid \varepsilon$ ، آنگاه :
  $First(A) = First(\delta) + First(\alpha)$

واضح است که اگر B را تهي یا ε درنظر گرفته شود ، آنگاه قاعده $A \rightarrow B\,\alpha$ در واقع بصورت $A \rightarrow \alpha$ تبدیل میشود. بنابراین First(A) شامل First(α) نیز میشود.

- چنانچه براي A گسترشی بصورت $A \rightarrow X1\ X2\ X3\ \ldots\ Xn$ وجود داشته باشد وبراي X1 تا Xi قاعده تهي هم وجود داشته باشد یا به عبارت دیگر

$\forall\ 1 \le j \le i\ \ Xj \rightarrow \delta j \mid \varepsilon$

آنگاه :

$First(A) = First(X1) + First(X2) + \ldots + First(Xi+1)$

واضح است که در قاعده $A \rightarrow X1\ X2\ X3\ \ldots\ Xn$ اگر X1 وجود داشته باشد ، آنگاه

$First(A) = First(X1)$

وگرنه در صورتیکه X1 تهي یا ε باشد ، آنگاه قاعده بصورت زیر خواهد بود

$A \rightarrow X2\ X3\ \ldots\ Xn$

به این ترتیب :

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

حالا اگر X1 و X2 هر دو تهي باشند آنگاه

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

به همين ترتيب مي توان نهايتا" نشان داد كه :

$$First(A) = \sum First(Xj) , 1 \le j \le I+1$$

چنانچه براي ترم A گسترش تهي نيز وجود داشته باشد يا بعبارت ديگر در حالت كلي قواعد مربوط به ترم A بصورت $A \rightarrow \alpha \mid \varepsilon$ باشد ، آنگاه با مشاهده يك عنصر متعلق به First(α) واضح است كه گسترش $A \rightarrow \alpha$ بايد انتخاب شود. اما چه عنصري بايد ظاهر شود تا در حاليكه انتظار مشاهده A ميرود ، بتوان دريافت كه گسترش $A \rightarrow \varepsilon$ بايد انتخاب شود. واضح است كه اگر ترم مورد انتظار يعني A در ورودي ظاهر نشود بايد ترمي كه پس از A انتظار مشاهده آن در ورودي ميرفت ظاهر شود. براي نمونه به گرامر زير توجه كنيد :

LabeledSt $\rightarrow$ Label Statement

Label $\rightarrow$ No : | ε

Statement $\rightarrow$ AssignmentSt | IfSt | WhileSt

AssignmentSt $\rightarrow$ Id :=
Expression

WhileSt $\rightarrow$ WHILE Expression DO Statement

IfSt $\rightarrow$ IF Expression DO Statement

طبق گرامر فوق جملات داراي برچسب يا Label و بدون برچسب هستند. در آغاز كار طبق گرامر فوق جهت مشاهده LabeledSt ابتدا انتظار مشاهده يك Label در ورودي ميرود. بنابراين انتظار ميرود كه اولين ترم در ورودي يك عدد يا No باشد زيرا :

No $\in$ First(Label) = { No,
ε)

اما ، اگر ترم خوانده شده از ورودي از نوع No نباشد ، مشخص است كه Label در ورودي وجود نداشته است. بتابراين ، انتظار ميرود كه ترم خوانده شده آغاز كننده ترم بعد از Label يعني Statement باشد. بنابراين چنانچه ترم مورد انتظار داراي گسترش تهي هم باشد ، درصورتي گسترش تهي انتخاب ميشود كه ترم پيش بيني متعلق به مجموعه First ترم بعدي آن در سمت راست قاعده مورد نظر باشد. مجموعه First ترمهايي كه پس از ترم مياني A در سمت راست قواعد متفاوت ظاهر ميشوند را در اصطلاح مجموعه پيرو يا Follow براي آن ترم مياني مي‌گويند. براي نمونه در گرامر فوق :

Follow(Label) = First(statement) = {S_Id, S_If, S_While}

اصولا" براي بدست آوردن مجموعه پيرو به روشهاي زير عمل ميشود:

- در صورتيكه ترم A در سمت راست يك قاعده بصورت زير ظاهر شود

$B \rightarrow A\,\alpha$

آنگاه :

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

- در صورتيكه ترم A در سمت راست يك قاعده بصورت زير ظاهر شود

$B \rightarrow \alpha$

A

آنگاه مجموعه پيرو براي A شامل مجموعه پيرو B ميباشد

$Follow(A) \supseteq Follow(B)$

اما ، بالعكس صادق نيست . علت اين است كه هر جايي كه B درسمت راست قواعد ظاهر شود ميتوان بجاي آن $A\,\alpha$ را قرار داد اما ، بالعكس صادق نيست.

- همواره در مجموعه پيرو براي سرترم گرامر علامت خاتمه فايل يا End Of File وجود دارد. زيرا ورودي تحليلگر نحوي يك فايل است. براي نمونه يك برنامه C را در نظر بگيريد. اين برنامه بعنوان يك جمله ورودي به كامپايلر داده ميشود. چون برنامه در فايل است در انتهاي آن علامت خاتمه فايل قرار ميگيرد. برنامه ورودي در اينجا از سرترم گرامر يعني برنامه C مشتق ميشود. پس بعد از سرترم گرامر علامت خاتمه فايل قرار ميگيرد. بطور كلي هر جمله اي كه در ورودي كامپايلر قرار ميگيرد بايد از سرترم گرامر مربوطه مشتق شود. چون جمله ورودي در داخل يك فايل است لذا ، همواره در مجموعه پيرو سرترم گرامرعلامت خاتمه فايل وجود دارد. علامت خاتمه فايل را بطور اختصاري معمولاً با $ مشخص ميكنند.

بنابراين مشاهده ميكنيد چنانچه مجموعه First براي يك ترم شامل عنصر تهي ε باشد ، آنگاه بايد عنصر ε را از داخل مجموعه حذف و بجاي آن عناصر مجموعه پيرو را به داخل مجموعه First افزود . به اين ترتيب :

$First(Label) = \{ No, \varepsilon \} = First(Label) + Follow(Label)$
$= \{ No, \{S\_Id,\ S\_If,\ S\_While\}$

مشاهده ميكنيد مجموعه Follow براي يك ترم شاخص گسترش تهي براي آن ترم است. بنابراين ميتوان نتيجه گرفت :

يك گرامر LL(1) است اگر اشتراك مجموعه سرآغاز براي هر دو گسترش هر ترم متعلق به آن گرامر تهي باشد و چنانچه آن ترم داراي گسترش تهي باشد اشتراك مجموعه سرآغاز و پيرو آن ترم بايد تهي باشد.

براي نمونه گرامر زير LL(1) نيست :

$LabeledSt \rightarrow Label\ Statement$

Label → Id : | ε

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}

بنابراین با مشاهده Id در ورودي نمي توان تصميم گرفت که آیا گسترش Label → Id : باید انتخاب شود یا گسترش Label → ε.

براي بدست آوردن مجموعه First میتوان از ماتریس تجانس نیز استفاده نمود. براي نمونه به گرامر زیر توجه کنید :

P → D B | B
B → A e
A → b S | S
S → a D | B a
D → a D | d

چنانچه در حالت کلي ترم A با ترم B آغاز شود يك رابطه بطول يك بين ترم A یا B وجود دارد. میتوان به این ترتیب يك گراف رواسط ایجاد نمود. اکنون مسإله بدست آوردن روابط بین هر دو گره در گراف است. یعني هدف بدست آوردن وجود روابط با هر طولي بین هر دو گره در داخل این گراف جهت دار روابط است. میتوان از ماتریس تجانس استفاده نمود. به این ترتیب که اگر بین دو ترم A و B يك رابطه آغاز شدن A توسط B وجود دارد در سطر مربوط به A و ستون مربوط به B در ماتریس مقدار يك قرار داده میشود. به این ترتیب براي گرامر فوق ماتریس زیر ایجاد میشود :

	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 ضرب باید نمود تا اینکه در دو تکرار متوالي مقادیر ماتریس با یکدیگر تفاوتي نکند . در اینصورت ماتریس نهایي روابط First را مشخص میکند. باید توجه داشته باشید که در هنگام ضرب دو ماتریس بولین

بجاي ضرب از عمل و منطقي يا AND استفاده ميشود و بجاي جمع عمل از OR منطقي استفاده ميشود.

٤.٦ تبديل گرامرها به فرم LL(١)

همانگونه كه قبلا نيز توضيح داده شد اگر هر دو گسترش متفاوت هر ترم متعلق به يك گرامر نهايتاً با يك ترم مشترك آغاز نشوند آن گرامر LL(1) است . براي تبديل گرامر به فرم LL(1) بعضاً ميتوان از روشي موسوم به *فاكتوگيري* چپ استفاده نمود .

4.6.1 - فاكتورگيري چپ

چنانچه در حالت كلي براي ترم A گسترش‌هايي بصورت زير موجود باشد :

$$A \rightarrow a\,\alpha \mid a\,\beta \mid \gamma$$

آنگاه :

$$first(a\,\alpha) \cap first(a\,\beta) = \{ a \}$$

با فاكتورگيري a از سمت چپ دو گسترش aβ و aα ميتوان گرامر را بصورت زير بازسازي نمود :

$$A \rightarrow a\,B \mid \gamma$$

$$B \rightarrow \alpha \mid \beta$$

در فرم توسعه يافته ميتوان بدون استفاده از ترم كمكي B عمل فاكتور گيري را انجام داد و گسترش‌هاي ترم مياني A را بصورت زير تبديل نمود:

$$A \rightarrow a\,(\alpha \mid \beta) \mid \gamma$$

با استفاده از روش فاكتور گيري چپ ميتوان قواعد مربوط به سرترم S از گرامر ارائه شده در بخش قبلي را بصورت زير بفرم LL(1) تبديل نمود :

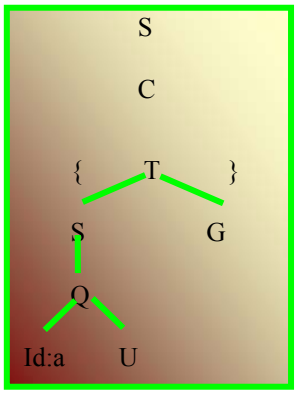
$$S \rightarrow I \mid W \mid Q \mid C$$

$$Q \rightarrow id\ U$$

$$U \rightarrow := No \mid \text{‘(‘ M ‘)’}$$

به اين ترتيب اكنون ميتوان بكار توليد درخت تجزيه كه در بخش قبل با مشاهده id يعني a در ورودي متوقف گرديد ادامه داد زيرا ، حالا با ديدن ترم پاياني **a** در

ورودي بلافاصله گسترش $Q \rightarrow S$ و پس از آن گسترش $Q \rightarrow id\ U$ انتخاب ميشود و درخت تجزيه به اين صورت تبديل ميگردد



حالا با فرا خواني تحليلگر لغوي در ورودي = : ظاهر ميشود كه در اينجا بطور قطعي ميتوان گفت كه گسترش $U \rightarrow :=No$ بايد انتخاب شود. در حالت كلي اگر در گرامر قواعد بصورت خود بازگشتي چپ وجود داشته باشد ، باز هم گرامر LL (1) نميتواند باشد.

4.6.2- تبديل قواعد خود بازگشتي چپ

وجود قواعد بصورت خود بازگشتي چپ مغاير با **LL (1)** بودن گرامرها ميباشد . براي نمونه به قواعد زير توجه نماييد :

$$A \rightarrow A\,\alpha \mid \beta$$

جهت نقض **LL (1)** بودن گرامر كافي است اثبات شود كه :

$$first(A\,\alpha) \cap first(\beta) \neq \varnothing$$

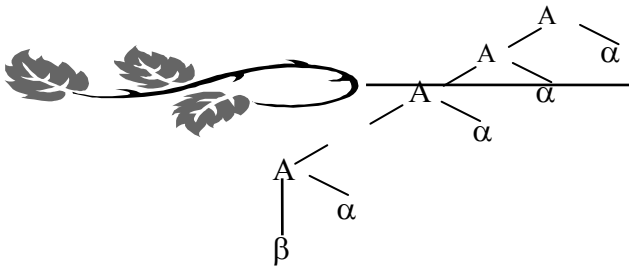
در اينجا واضح است كه :

$first(A\,\alpha) \cap first(\beta) = \beta$ چرا

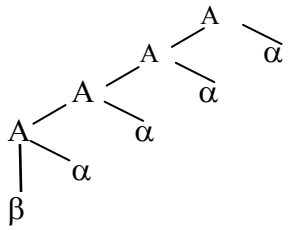
كه :

$$first(A\,\alpha) = first(A) = First(\beta)$$

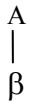
براي تبديل قواعد مربوط به A بصورت LL(1) به اين ترتيب ميتوان استدلال نمود كه دو قاعده $A \rightarrow A\,\alpha \mid \beta$ نمايانگر فرم كلي رشته هائي است كه با β آغاز و با تعداد صفر يا بيشتر α ادامه مي يابند. بنابراين هر رشته با فرم كلي $\beta\alpha\alpha\alpha$ ..... α از سر ترم A بايد مشتق شود.



به این ترتیب براي رشته β α α α درخت تجزیه بصورت زیر خواهد بود :



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

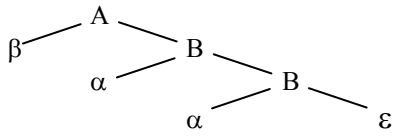


رشته هاي فوق را با گرامر زیر نیز میتوان تولید نمود:

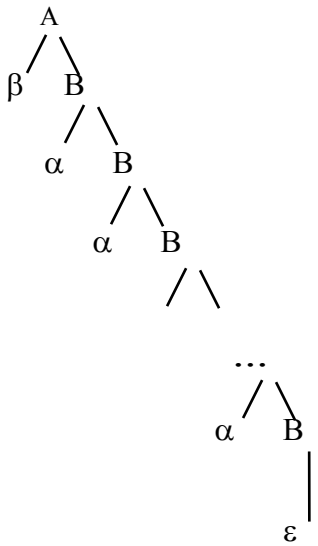
$A \rightarrow \beta B$

$B \rightarrow \alpha B \mid \varepsilon$

رشته β α α α را در نظر بگیرید. این رشته بر طبق گرامر فوق از سر ترم A مشتق میشود :



و به همین ترتیب هر رشته با فرم کلي α..... βααα از سر ترم A باید مشتق مي شود. درخت تجزیه در حالت کلي بصورت زیر است.



بنابراين ميتوان نتيجه گرفت كه براي حذف خودبازگشتي چپ از قواعد با فرم كلي :

$A \rightarrow A\,\alpha \mid \beta$

مي توان آنها را با قواعد معادل زير جايگزين نمود :

$A \rightarrow \beta\,B$

$B \rightarrow \alpha\,B \mid \varepsilon$

در فرم توسعه يافته گرامر بصورت زير تبديل ميشود :

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

در فرم توسعه يافته آكولاد به مفهوم تكرار صفر يا بيشتر است.

مثال گرامر عبارات را بصورت **LL (1)** تبديل كنيد .

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow Id \mid No \mid (\,E\,)$

به قواعد E توجه نماييد. دو گسترش $E + T$ و $E - T$ با يك ترم مشترك E آغازميشوند. به همين ترتيب دو گسترش متفاوت ترم مياني T نيز با يك ترم آغاز ميشوند . پس از انجام عمل فاكتورگيري چپ گرامر توسعه يافته عبارات بصورت زير تبديل ميشود:

$E \rightarrow E\,(+ \mid -)\,T \mid T$

$T \rightarrow T\,(* \mid /)\,F \mid F$

$F \rightarrow id \mid NO \mid (\,E\,)$

حالا با حذف خود بازگشتي چپ ، گرامر به صورت زير تبديل مي شود :

$E \rightarrow T\,\{\,(+ \mid -)\,T\,\}$

$T \rightarrow F\,\{\,(* \mid /)\,F\,\}$

$F \rightarrow Id \mid NO \mid (\,E\,)$

درصورت وجود قواعد تهي با فرم كلي $A \rightarrow \varepsilon$ درگرامر نيز ممكن است گرامر LL(1) نباشد.

4.6.3- حذف قواعد تهي

چنانچه در گرامري قواعد تهي وجود داشته باشد ، آن گرامر ممكن است LL(1) نباشد. براي تبديل گرامر به فرم LL(1) بايد ابتدا قواعد تهي را حذف نمود و سپس

با استفاده از روشهاي فاكتور گيري و حذف خودبازگشتي چپ گرامر را به فرم LL(1) تبديل نمود. براي نمونه به گرامر زير توجه نمانيد :

LabeledSt ⟶ Label Statement

Label ⟶ Id : | ε

Statement ⟶ AssignmentSt | CallSt

AssignmentSt ⟶ Id := Expression

CallSt ⟶ Id '(' ActualParams ')' | Id

ActualParams ⟶ Expression | Expression , Params

Expression ⟶ Id | No

در گرامر فوق براي ترم مياني Label يك گسترش تهي وجود دارد. لذا بايد First (Label) ∩ Follow(label) = ∅ باشد. اما :

First(label) = { Id }

و

Follow(label) = First(Statement) = First(AssignmentSt) + First(CallSt) = { Id }

بنابراين :

First (Label) ∩ Follow(label) = { Id } ≠ ∅

بنابراين گرامر LL(1) نيست زيرا هنگاميكه انتظار مشاهده Label در ورودي مي رود ، با مشاهده Id در ورودي نمي توان تصميم گرفت كه آيا گسترش Label ⟶ Id : بايد انتخاب شود و يا گسترش Label ⟶ ε .

براي تبديل گرامر فوق به فرم LL(1) بايد گسترش Label ⟶ ε حذف شود و هر جايي كه از تزم مياني Label در سمت چپ يك قاعده استفاده شده است ، يك بار Label را تهي و بار ديگر بصورت غير تهي در نظر گرفت. بنابراين گرامر بصورت زير تبديل ميشود :

LabeledSt ⟶ Statement | Label Statement

Label ⟶ Id :

براي LabeledSt اكنون First(statement) ∩ Follow(label) = {Id} ≠ ∅ پس بايد با استفاده از عمل فاكتور گيري چپ گرامر را به فرم LL(1) تبذيل نمود. براي اين منظور بايد ابتدا ترم ها را جايگزين كرد .

LabeledSt ⟶ Id := Expression | Id '(' ActualParams ')' | Id : Statement

پس از فاکتورگیري از Id قاعده بصورت زیر تبدیل میشود :

LabeledSt $\longrightarrow$ Id

St

St $\longrightarrow$:= Expression | '(' ActualParams ')' | : Statement

در مورد Statement نیز باید عمل فاکتور گیري چپ را انجام داد:

Statement $\longrightarrow$ AssignmentSt | CallSt

با جایگزیني AssignmentSt و Callst قاعده مربوط به Statement بصورت زیر تبدیل میشود:

Statement $\longrightarrow$ Id := Expression | Id '(' ActualParams ')'

اکنون با استفاده از عمل فاکتورگیري چپ قاعده فوق بصورت زیر تبدیل میشود :

Statement $\longrightarrow$ Id StTail

StTail $\longrightarrow$:= Expression | Id '(' ActualParams ')'

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

S $\longrightarrow$ L A B

L $\longrightarrow$ d | ε

A $\longrightarrow$ d A | B a

B $\longrightarrow$ B b | ε

گرامر فوق LL(1) نیست زیرا :

First (L) $\cap$ Follow(L) = { d } $\neq \varnothing$

بنابراین باید گسترش تهي $L \longrightarrow \varepsilon$ حذف شود. در اینصورت باید در هر قاعده اي که L ظاهر میشود یك بار مقدار آنرا تهي و یك بار غیر تهي در نظر گرفت. بنابراین گرامر بصورت زیر تبدیل میشود :

S $\longrightarrow$ A B | L A B

L $\longrightarrow$ d

A $\longrightarrow$ d A | B a

B $\longrightarrow$ b B | ε

توجه داشته باشید که پس از حذف خود بازگشتي چپ قواعد $B \longrightarrow B\ b \mid \varepsilon$ را مي توان بصورت $B \longrightarrow b\ B \mid \varepsilon$ ساده نمود. البته هنوز گرامر فوق LL(1) نیست زیرا اشتراك مجموعه سرآغاز دو گسترش مختلف ترم میاني S تهي نمي باشد.

First (A B) $\cap$ Follow(L A B) = { d } $\neq \varnothing$

بنابراین باید با جایگزیني ترمهاي A و L در آغاز دو گسترش ترم S و سپس عمل فاکتور گیري چپ گرامر را LL(1) نمود.

S $\longrightarrow$ d A B | B a B | d A B

A $\longrightarrow$ d A | B a

$B \longrightarrow b\ B \mid \varepsilon$

در نتیجه گرامر بصورت زیر تبدیل میشود :

$S \longrightarrow d\ A\ B \mid B\ a\ B$

$A \longrightarrow d\ A \mid B\ a$

$B \longrightarrow b\ B \mid \varepsilon$

٤.٦ ایجاد جدول تجزیه بالا به پائین

جدول تجزیه ساختاري براي توليد برنامه تحليلگر نحوي براي گرامرهاي LL(1) است. براي نمونه به گرامر زير توجه نمائيد.

$S \longrightarrow d\ A\ B \mid B\ a\ B$

$A \longrightarrow d\ A \mid B\ a$

$B \longrightarrow b\ B \mid \varepsilon$

براي ايجاد تجزيه گر براي اين گرامر بايد ابتدا مجموعه هاي سرآغاز را براي ترمهاي مياني بدست آورد. با استفاده از اين مجموعه ها جدول تجزيه بدست مي آيد :

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

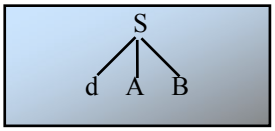
	a	b	d	$
S	BaB	BaB	dAB	
A	Ba	Ba	dA	
B	ε	bB		ε

اکنون با استفاده از جدول تجزيه فوق ميتوان عمل تجزيه بالا به پائين را انجام داد. براي نمونه جمله ddabbb را با استفاده از جدول تجزيه فوق ميتوان براحتي تجزيه نمود. براي اين منظور از يك پشته بنام پشته تجزيه استفاده ميشود.

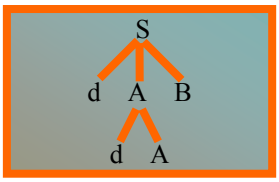
پشته	ورودي	قاعده تجزيه
$S	ddabbb$	$S \longrightarrow d\ A\ B$
$B A d	ddabbb$	
$B A	dabbb$	$A \longrightarrow d\ A$
$B A d	dabbb$	
$B A	abbb$	$A \longrightarrow B\ a$
$B a B	abbb$	$B \longrightarrow \varepsilon$
$ a B	abbb$	
$B	bbb$	$B \longrightarrow bB$
$bB	bbb$	
$B	bb$	$B \longrightarrow bB$
$bB	b b$	
$B	b $	$B \longrightarrow bB$
$bB	b$	

همانگونه که در جدول فوق مشاهده میکنید به انتهاي رشته ورودي علامت خاتمه فایل یعني $ افزوده میشود. عمل تجزیه از سرترم گرامر آغاز میشود. و پس از سرترم انتظار میرود که در ورودي علامت خاتمه فایل ورودي یعني $ ظاهر شود. لذا بنار این پیش بیني در آغاز کار رشته $S در داخل پشته قرار داده شده است.

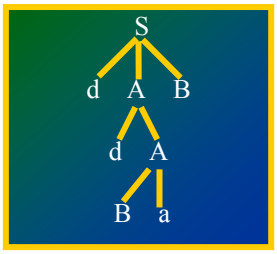
عمل تجزیه از ردیف مربوط به سرترم گرامر S آغاز میشود. با مشاهده اولین ترم در جمله **d**dabbb$ که ترم در ورودي که ترم پایانی d است به ستون d در سطر S در داخل جدول ارجاع میشود. گسترش BaB در مکان (S,d) از جدول مشخص شده است. لذا، درخت تجزیه بصورت زیر ایجاد میشود:



اکنون، ترم بعدي ازجمله dd**a**bbb$ خوانده میشود. این ترم پیش بیني نیز d است. با توجه به اینکه ترم پیش بیني هنوز d است. طبق درخت تجزیه انتظار مشاهده A در وردي میرود. درمکان (A,d) از جدول تجزیه گسترش dA قرار دارد. درخت تجزیه زیر حاصل میشود.

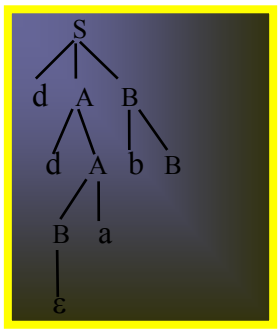


ورودي بعدي در جمله dd**a**bbb$ ترم پایانی a است. طبق درخت تجزیه انظار مشاهده A در ورودي میرود. درمکان (A,a) از جدول گسترش Ba قرار گرفته است. بنابراین درخت تجزیه بصورت زیر توسعه داده میشود :

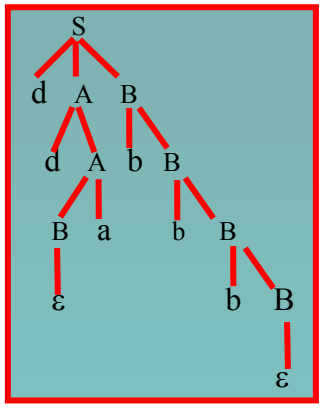


حالا با توجه به محتوي جدول تجزیه در مکان (B,a) باید گسترش B → ε انتخاب شود. بنابراین طبق درخت تجزیه انتظار دیدن a در ورودي میرود. در این لحظه ترم پیش بیني همان طوري که انتظار میرود a است. ورودي بعدي در جمله dda**b**bb ترم پایانی b است. طبق درخت تجزیه انظار مشاهده B در ورودي میرود. درمکان

(B,b) از جدول گسترش bB قرار گرفته است. بنابراین تا این مرحله درخت تجزیه بصورت زیر مي باشد:



به همین ترتیب اگر عملیات ادامه پیدا کند تهایتاً درخت تجزیه بصورت زیر ایجاد میشود :



مثال - گرامر عبارات را که در زیر ارائه شده تبدیل به فرم LL(1) نموده ، جدول تجزیه براي يك تحليگر پيش بيني كننده ايجاد كنيد:

Expression ⟶ Expression RelOp SimpleExp | SimpleExp

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

VarTale

VarTale ⟶ '[' Dim ']' .Variable |

∈

Dim ⟶ Expression

OtherDims

OtherDims ⟶ ' ,' Dim | ∈

قواعده ارائه شده براي Expression داراي وضعيت خودبازگشتي چپ مستقيم است.

Expression ⟶ Expression RelOp SimpleExp | SimpleExp

درحالت كلي $A \rightarrow A\alpha \mid \beta$ را مي توان با قواعد معادل زير جايگزين نمود :

$A \rightarrow \beta B$

$B \rightarrow \alpha B \mid \varepsilon$

حالا Expression را مشابه A و RelOp SimpleExp مشابه با α و SimpleExp را مشابه با β در نظر بگیرید. به این ترتیب قواعد مربوط به E بصورت زیر تبدیل میشوند :

Expression $\rightarrow$ SimpleExp E

E $\rightarrow$ RelOp SimpleExp E | ε

در مورد SimpleExp ابتدا باید فاکتورگیري چپ نمود .

SimpleExp $\rightarrow$ SimpleExp '+' Term | SimpleExp '-' Term | SimpleExp Or Term | Term

پس از فاکتورگیري چپ قواعد بصورت زیر تبدیل میشوند :

SimpleExp $\rightarrow$ SimpleExp Simple | Term

Simple $\rightarrow$ '+' Term | '-' Term | Or Term

پس از حذف خودبازگشتي چپ قواعد به صورت زیر تبدیل میشوند :

SimpleExp $\rightarrow$ Term S

S $\rightarrow$ Simple S | ε

Simple $\rightarrow$ '+' Term | '-' Term | Or Term

درقاعده مربوط به S میتوان Simple را با گسترش آن جایگزین نمود. بنابراین قواعد فوق بصورت زیر ساده میشود :

SimpleExp $\rightarrow$ Term S

S $\rightarrow$ '+' Term S | '-' Term S | Or Term S | ε

بطورخلاصه گرامر عبارات در فرم LL(1) بصورت زیر مي باشد:

Expression $\rightarrow$ SimpleExp E

E $\rightarrow$ RelOp SimpleExp E | ε

RelOp $\rightarrow$ < | <= | = | <> | >= | > | IN

SimpleExp $\rightarrow$ Term S

S $\rightarrow$ '+' Term S | '-' Term S | Or Term S | ε

Term $\rightarrow$ Factor T

T $\rightarrow$ '/' Factor T | '*' Factor T | DIV Factor T | AND Factor T | ε

Factor $\rightarrow$ Number | NOT Factor | '(' Expression ')' | Variable

Variable → Identifier

VarTale

VarTale → '[' Dim ']' .Variable |

ε

Dim → Expression

OtherDims

OtherDims → ' ,' Dim | ε

اکنون میتوان مجموعه سرآغاز را براي ترمها محاسبه نمود :

First(Expression) = First(SimpleExp) = First(Term) = First(factor) = {Number, NOT, (, Identifier}

First(E) = First(RelOp) + {∈} = { < , <= , = , <> , >= , > , IN , ε }

Follw(E) = Follow(Expression) = First(OtherDims) + { ')' , $

}

First(OtherDims) = {',', ε } = {','} + Follow(OtherDims) = = {','} + Follow(Dim) = { ',' , ']' }

First(E) = { < , <= , = , <> , >= , > , IN } + { ')' , $, ',' , ']' }

First(S) = { + , - , OR , ε } = { + , - , OR } +

Follow(S)

Follow(S) = Follow(SimpleExp) =

First(E)

First(S) = { + , - , OR } + { < , <= , = , <> , >= , > , IN , ')' , $, ',' , ']' }

First(T) = { / , *, DIV, AND, ε } = { / , *, DIV, AND} + Follow(T)

Follow(T) = Follow(Term) =

First(S)

First(T) = { / , *, DIV, AND} + {+ , - , OR , < , <= , = , <> , >= , > , IN , ')' , $, ',' , ']'}

Fisr(Variable) =

{Identifier}

First(VarTale) = {'[' } +

Follow(VarTale)

Follow(VarTale) = Follow(Variable) = Follow(factor) =

First(T)

First(Variable) = {'['}+{ /, *,DIV, AND,+ ,-,OR ,< ,<= ,=,<>,>=,>,IN,')',$,',',']'}

First(Dim) = First(Expression) = {Number, NOT, (,

Identifier}

حالا با استفاده از روابط فوق مي توان جدول تجزیه بالا به پائین را بسادگي ایجاد کرد.

مثال - گرامر زیر را به فرم LL(1) تبدیل نموده ، جدول تجزیه براي آن ایجاد کنید.

S → SA |

BdA

A → Aa

| b

B → ba | Bd |

ε

تبدیل گرامر به فرم LL(1) در فرم توسعه یافته بسیار ساده تر است. لذا، گرامر بصورت زیر تبدیل میشود.

S → BdA {A}
A → b {a
}
B → [ba] {d
}

اما چون First (B) ∩ Follow(B) = { d } ≠ ∅ گرامر LL(1) نمي باشد. بنابراين بصورت زير عمل بايد نمود :

S → [ba] {d }dA {A}
A → b {a
}

درمورد گسترش S ، مشکل {d }d است. زيرا مشخص نيست که چه هنگامي مي توان به d دومي رسيد. اما ، واضح است که {d }d = d{d } است. بنابراين گرامر بصورت زير تبديل ميشود :

S → [ba] d{d }A {A}
A → b {a
}

حالا ميتوان جهت توليد جدول تجزيه گرامر را به فرم ساده تبديل نمود. براي اين منظور ابتدا [b a] را با B ، d {d} را با C و A{A} را با D بايد جايگزين نمود.

S → B C D
A → b F

→ a F | ε
F

B → b E | a | ε
E → a | ε
C → d C | d
D → A D | ε

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

First(F) = { a , ε } Follow(F) = Follow(A) = First(D) = {b}
First(B) = {b , a, ε} Follow(B) First(C) = {d}
=

First(E) = { a, ε} Follow(E) = Follow(B) = {d}
First(C)
={d}

First(G) = { d , ε } Follow(G) = Follow(C) = First(D) = {b}
FIrst(D) = { b, ε } Follow(D) = Follow(S) = {$}

اکنون مي توان جدول تجزيه را بسادگي براي اين گرامر بدست آورد.

٤.٧ تجزیه گرهای کاهینه بازگشتی

تجزيه گرهاي *كاهينه بازگشتي* يا Recursive scent parser را مي توان براي گرامرهاي LL(1) مورد استفاده قرار داد. در اين روش براي هر ترم مياني و سرترم ، يك روال يا زير برنامه به همان نام ايجاد ميشود. به اين ترتيب ، قواعد گرامر را عيناً با استفاده از توابع خود بازگشتي تبديل به كد برنامه مينمايند. لذا، مزيت اين روش سادگي ايجاد وخوانايي كد برنامه تجزيه گر است.

در ساختاركلي يا در واقع برنامه اصلي براي اين نوع تحليلگر پس از انجام عمليات اوليه بنام Init تحليلگر لغوي بنام NextSymbol فراخواني فراخوانده ميشود تا نوع اولين ترم پيش بيني در يك متغير سرارسري بنام CurrentSymbol قرار گيرد. سپس روال تعيين شده براي سرترم گرامر كه در زير ProgramX ناميده شده ، مورد فراخواني قرار ميگيرد. بنابراين بدنه اصلي برنامه تحليلگر كاهينه بازگشتي بصورت زير است :

```
Begin
   init;
   NextSymbol;

                                                    ProgramX

End .
```

Current Symbol متغيري سراسري از نوع شمارش پذير Symbols است كه قبل از اين براي تعيين نوع لغات توسط تحليلگر لغوي مشخص شد. نوع Symbols حاوي انواع لغات موجود در زبان مورد نظر است. براي 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_lt , 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 اينكه تحليلگرلغوي اولين لغت را از برنامه ورودي تشخيص و در داخل CurrentSymbol قرار داد ، ميبايست روال ProgramX فراخواني شود. در واقع ProgramX نام سر ترم گرامر است چرا كه در اين روش براي هر ترم مياني و سر ترم روالي به همان نام نوشته ميشود. روال ProgramX بر اساس قاعده مربوطه نوشته شده است. قاعده مربوط به ProgramX در زير با يك جمله تفسيري كامنت مشخص شده است:

```
(* Program  X  →   Program id ';' BlockBody  '.' * )
```

```
Procedure        } روال تشخیص سرترم گرامر
                                                    ProgramX ; {
    Begin
      Expect ( S_Program ) ;   { انتظار مشاهده S_Program در ورودي ميرود }
       Expect ( S_ id ) ;    { انتظار مشاهده S_Id در ورودي ميرود }
      Expect ( S_Semicolon ) ; { انتظار مشاهده S_Semicolon در ورودي ميرود }
BlockBody ;  {فراخواني روال ترم مياني BlockBody براي تشخيص بدنه برنامه
                                                                    {
      Expect ( S_dot ) { انتظار مشاهده S_Semicolon در ورودي ميرود }
    End ;
```

همانطوریکه مشاهده میشود اکنون در ابتداي روال ProgramX بنا بر قاعده مربوطه ، انتظار مشاهده لغت از نوع S_Prgram میرود. براي اين منظور تابع Expect با پارامتر S_Program مورد فراخواني قرار گرفته است. ترم پيش بيني قبل از اجراء دستورالعمل Expect (S_Program) در داخل برنامه اصلي و با فراخواني NextSymbol در داخل CurrentSymbol قرار داده شد. بنابراين محتوي CurrentSymbol در داخل Expect بنابرگرامر با S_Program مقايسه ميشود. کد اين روال بصورت زير است :

```
  Procedure EXPECT ( S : Symbols ) ;
  Bigin
If CurrentSymbol = S    { اگر ترم پيش بيني مساوي با پارامتر ارسالي است
                                                                    {
Then  NextSymbol        { آنگاه لغت بعدي بعنوان ترم پيش بيني خوانده شود
                                                                    {
   Else  SyntaxError    { وگرنه پيام خطا نحوي صادر شود }
   End ;
```

اکنون مي توان روال مربوط به ترم مياني BlockBody را نوشت. در ابتدا بايد قواعد مربوط به اين ترم مياني را مشخص نمود تا بتوان بر اساس آن گرامر روال مربوطه را مشخص کرد :

```
( * Blockbody → [ ConstantDef.part ]
                  [ typeDef.Part ]
                 [VarDefPart ]
                  {FunctionDef | ProcedureDef }
                  Compaund Statement * )
Procedure  BlockBody ;
          Begin
        if  CurrentSymbol =  S_const
            then  ConstantDef.Part ;
        if  CurrentSymbol =  S_type
           then  TypeDefPart ;
        if  CurrentSymbol =  S_Var
                  then  VarDefPart  ;
          While ( current Symbol =  S_ Procedure  |  Current symbol =  S_Function  )
                do if  CurrentSymbol  =  S_Procedure
                       then  ProcedureDef.
                     else  FunctionDef.
          Compound Statement  ;
    End ;
```

درگرامرفوق براكت علامت تكرار صفر و يا يك و آكولاد علامت تكرار صفر يا بيشتر است. نكته قابل توجه اين است كه در داخل ProgramX هنگاميكه BlockBody فراخواني ميشود . لغت بعدي در داخل CurrentSymbol قرار دارد. لذا، اگر اين لغت S_Const باشد روال مربوط به ثابتها فراخواني ميشود. اين روال بخش تعريف ثابتها را بنا بر گرامر ، مورد تحليل نحوي قرار ميدهد و در خاتمه لغت بعدي را در داخل CurrentSymbol قرار ميدهد . همين امر موجب ميشود كه تحليلگر نحوي بتواند بسادگي به كار خود ادامه دهد.

نكته قابل توجه در مورد تشخيص روالها و توابع است . به هر تعدادي و يا هر تركيبي از آنها را ميتوان داشت لذا در داخل يك حلقه While پس از اينكه يك روال يا تابع تشخيص داده ميشود بايد مطمئن بود كه لغت بعدي در CurrentSymbol است كه اگر S_Procedure يا S_Function باشد در داخل حلقه مربوط فراخواني ميشود :

```
  ( * ConstantDefPart → Const  ConstantDef {ConstantDef } *)
Procedure                Part;
                                                    ConstantDef
       Bigin
     Nextsymbol ;
          ConstantDef ;
      While  CurrentSymbol = S_id
           Do ConstantDef ;
End ;
```

```
( * constant Def  → id '='  ( No | id ) ' ; '  *
                                                              )
Procedure  ConstantDef
                                                              ;
        Begin
    Expect ( S_id ) ;
    Expect ( S_EQ ) ;
     If  CurrentSymbol  =  S_No
            Then  Nextsymbol
            else  Expect ( S_id ) ;
     Expect ( S_Semicolon ) ;
End ;
```

در بخش قبلي گرامر عبارات به برم LL(1) تبديل شد تا بتوان جدول تجزيه براي تحليلگر نحوي پيش بيني كننده براي تشخيص عبارات ايجاد نمود. براي ايجاد تحليلگر كاهينه بازگشتي بهتر است كه گرامر در فرم توسعه يافته بصورت LL(1) تبديل شود. گرامر عبارات در فرم توسعه يافته بصورت زير است :

Expression → SimpleExp {RelOp SimpleExp}

RelOp → < | <= | = | <> | >= | > | IN

SimpleExp → Term { ('+' | '-' | Or) Term }

Term → Factor { ('/' | '*' | DIV | AND) Factor}

Factor → Number | NOT Factor | '(' Expression ')' | Variable

Variable → Identifier { '[' Dim ']' }

Dim → Expression { ‘ ,’ Expression }

حالا میتوان بر اساس گرامر فوق تجزیه گر کاهینه بازگشتي را بصورت زیر ایجاد کرد :

```
Begin   Init;   NextSymbol; Expression;
                                                                End.

(*Expression      →    SimpleExp    {RelOp
                                                              SimpleExp}*)
Procedure
                                                              Expression;

                                                                Begin

                                                              SimpleExp;
  while CurrentSymbol in First(RelOp)
                                                                  do

        Begin RelOp ; SimpleExp  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;  Term
                                                                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;
```

٤.٨ بهبود از خطاء

يكي از دامنه هاي تحقيقاتي بسيار وسيع در زمينه كامپايلر ها كه حتي امروزه به هوش مصنوعي نيز ارتباط پيدا كرده است ، مسئله بهبود از خطا است . مسئله اينجا است كه يك كامپايلر قادر باشد پس از مشاهده خطاي نحوي در متن برنامه داده شده به درستي بكار خود ادامه دهد و حد اكثر تعداد خطارا در يك مرحله از كامپايل تشخيص دهد .

نكته اين است كه براي نمونه اگر اشتباها" در برنامه اي بجاي If برنامه نويس اشتباها" Uf وارد كند ، كامپايلر با مشاهده لغت Uf كه يك شناسه است به جاي If ، گمراه نشده و پيامهاي خطاي اضافي صادر نكند و قادر باشد كه در يك مرحله از كامپايل حد اقل تعداد ثيام لازم براي حد اكثر تعداد خطا را ايجاد نمايد .

براي درك بهتر مسإله بهبود از خطا، در زير يك مثال ارائه ميشود. براي اين منظور روالهاي زير را كه قبلاً جهت تشخيص ثابتها ارائه شده بود در نظر بگيريد :

```
(  *      →  Const  ConstantDef  {ConstantDef  }    *)
                                                          ConstantDefPart
Procedure  ConstantDef Part
                                                          ;

                                                          Bigin
  NextSymbol ;
        ConstantDef ;
            While  CurrentSymbol = S_id
```

```
Do  CostantDef ;

                                                                    End ;

( *     → id   '='   ( No | id ) ' ; '   * )
                                                                    ConstantDef
Procedure  ConstantDef
                                                                    ;

                                                                    Begin
   Expect ( S_id ) ;
   Expect ( S_EQ ) ;
         If  CurrentSymbol  =  S_No
       Then  Nextsymbol
       Else  Expect ( S_id ) ;
   Expect ( S_Semicolon ) ;
     End ;
```

اکنون به قطعه کد مقابل که داراي خطاء نحوي است توجه نمائید .

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

براي تجزیه قطعه کد فوق روال ConstantDef فراخواني میشود. در ضمن عمل تجزیه هنگامي که پس از ترم پایانی a در ورودي انتظار مشاهده S_EQ میرود، برخلاف انتظار لغت := از نوع S_Assigin ظاهر میشود بنابراین در داخل Expect روال SyntaxError فراخواني میشود. در اینجا نکته چگونگی عملکرد SyntaxError است . اگر SyntaxError بصورت زیر نوشته شود :

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

چون در این حالت NextSymbol در داخل Expect مورد فراخواني قرار نمي گیرد ، محتواي CurrentSymbol همان S_Assigin باقي 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 یعني ConstantDef میرسد. امّا ، تغییر نکردن مقدار ترم پیش بیني یعني مقدار CurrentSymbol موجب میشود که به غلط پیام خطاء دومي اعلام شود. به این ترتیب زماني که دستورالعمل Expect (S_No) اجراء میشود چون محتواي CorrentSymbol مساوي با S_No نبوده و مساوي با S_Asigin است لذا ، مجدداً به غلط پیام خطاي دیگري صادر میشود و به همین ترتیب پیامهاي

خطاي بعدي ، يكي پس از ديگري صادر مي شوند. اگر NextSymbol بصورت زير در داخل SyntaxError فراخواني شود :

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

آنگاه پس از اعلام پيام خطا ، لغت بعدي يعني عدد 5 كه از نوع S_No است از ورودي خوانده مي شد. به اين ترتيب با تغيير CurrentSymbol به S_No ، روال ConstantDefبه درستي ميتوانست بكار خود ادامه دهد. با وجود اين ، افزايش NextSymbol يا بعبارت ديگر چشم پوشي از مورد خطا كه در اينجا S_Assigin بود همواره مشكل گشا نيست. براي مثال در سطر بعدي يعني b = ; پس از تشخيص = يا S_EQ محتواي CurrentSymbol به غلط S_Semicolon خواهد بود. و در زماني كه دستورالعمل Expect (S_No) اجرا ميشود محتواي CurrentSymbol ، S_Semicolon است و در اينجا چشم پوشي از اين لغت يعني S_Semicolon موجب خطا ميشود و بهتر است كه NextSymbol از داخل روال SyntaxError حذف شود .

چه بايد كرد ؟ مشاهده نموديد كه در يك مورد وجود NextSymbol در داخل SyntaxError براي چشم پوشي از مورد خطاي نحوي و در واقع لغت غير قابل انتظار ، ضروري است و در مورد ديگر اين عمل غلط ميباشد . براي رفع اين مشكل در داخل هر قاعده براي هر ترم بطور مجزاء.مجموعه اي از ترمها كه پس از آن ترم در قاعده اي ظاهر مي شوند را به عنوان مجموعه STOP يا متوقف كننده در نظر گرفته مي شود. اكنون در داخل SyntaxError آنقدر ترمهاي بعدي خوانده شده و از آنها چشم پوشي مي شود تا طبق قاعده بتوان به يكي از ترمهاي بعدي مورد انتظار در داخل مجموعه Stop رسيد .

براي نمونه طبق گرامر در داخل ConstantDef پس از S_Id انتظار ديدن S_EQ و پس از آن انتظار S_Semicolon در ورودي ميرود. پس از S_Semicolon مسلماً بايد ترمهايي كه پس از ConstantDef ميتوانند ظاهر شوند ، قرار گيرند. اين ترمها شامل S_id كه شروع كننده ConstantDef ديگري است و همين طور ترمهايي كه بعد از خود ConstantDef.Part ميتوانند ظاهر شوند ، است. به اين ترتيب روالها را ميتوان بصورت زير باز نويسي كرد :

```
( *    →  Const     ConstantDef.  {ConstantDef }     *)
                                                  ConstantDefPart
   Procedure  ConstantDef Part  ( Stop : Set of Symbols ) ;

                                                       Bigin
  NextSymbol ;
   ConstantDef  ( [ S_id ] + Stop ) ;
   While  CurrentSymbol = S_id
```

```
        Do  CostantDef  ( [ S_id ] + Stop ) ;

                                                                End ;

( *   → Id '='   ( No | id ) ' ; '   * )
                                                        ConstantDef
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 کلي واضح است که براي محاسبه مجموعه Stop بصورت زير عمل ميشود :

الف - چنانچه : $E \rightarrow e_1\ e_2\ e_3\ \ldots\ e_n$ آنگاه :

$$Stop(e_i) = \sum_{i=1\ldots n-1} First(e_j)_{j=(i+1)\ldots n} + Stop(E)$$

$$Stop(e_n) = Stop(E)$$

ب - چنانچه : $E \rightarrow e_1 \mid e_2 \mid \ldots\ldots\ldots \mid e_n$ آنگاه :

$$Stop(e_i) = Stop(E) \qquad i = 1\ldots n$$

ج - چنانچه : $E \rightarrow \{ e_1 \}$ آنگاه :

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

به این ترتیب برنامه تحلیلگر لغوي که قبل از این نوشته شده بود بصورت زیر تبدیل میشود :

```
Begin
  Init ;   //   عملیات مقدماتي
Nextsymbol ;          //  گرفتن لغت بعدي از تحلیلگر لغوي
ProgramX ( [ S_EOF ]       //  انتظار سرترم و علامت خاتمه فایل پس از آن
                                                                    )

End.
```

همانگونه که مشاهده مي نمایید ، پس از سر ترم گرامر ، انتظار مشاهده علامت پایان فایل یا S_EOF مي رود زیرا ، هر جمله ورودي به کامپایلر در داخل یك فایل ظاهر میشود که با علامت خاتمه فایل انتهاي آن مشخص میشود. بنابراین پس از هر جمله ورودي علامت خاتمه فایل قرار میگیرد. جمله ورودي در صورت صحیح بودن از سرترم گرامر مشتق میشود. بنابراین پس از سرترم گرامر همواره انتظار مشاهده علامت خاتمه فایل میرود.

```
    (* ProgramX → Program id ';' CompoundSt '.' *)
       Procedure  ProgramX ( Stop : Set of  Symbols ) ;
 Begin
         Expect ( S_program , [ S_id , S_Semicolon ] + First ( Blockbody ) + [ S_dot ] + Stop ) ;
        Expect ( S_id , [ S_Semicolon , First ( Blockbody ) + [ S_dot ] + Stop ) ;
       Expect ( S_Semicolon , First ( Blockbody ) + [ S_dot ] + Stop  ) ;
      Blockbody ( [ S_dot ] + Stop ) ;
     Expect ( S_dot , Stop ) ;
End;
   ( *  Blockbody  →  [ ConstantDef.part ][ typeDef.Part ][VarDefPart ]
                        {FunctionDef  | ProcedureDef  } Compaund Statement * )
  Procedure  BlockBody ( Stop : Set of  Symbols);
        Begin
        if  CurrentSymbol =  S_const
        then  ConstantDef.Part(Stop + [S_Type, S_Var, S_Procedure, S_Function, S_Begin] );
        if  CurrentSymbol =  S_type
       then  TypeDefPart( Stop + [ S_Var, S_Procedure, S_Function, S_Begin] );
   if  CurrentSymbol =  S_var
       then  VarDefPart( Stop + [ S_Procedure, S_Function, S_Begin] );
     while ( current Symbol =  S_ Procedure  |  Current symbol =  S_Function  )
          do if  CurrentSymbol  =  S_Procedure
                then  ProcedureDef ( Stop + [ S_Procedure, S_Function, S_Begin] )
                else  FunctionDef ( Stop + [ S_Procedure, S_Function, S_Begin] );
   Compound Statement( Stop ) ;
End ;
```

مثال - گرامر زیر را به فرم LL(1) تبدیل نموده ، یك تحلیلگر كاهینه بازگشتي براي آن ایجاد نمائید.

$S \rightarrow aB \mid aC \mid dD$

$D \rightarrow Da \mid Db \mid d$

$B \rightarrow BC \mid b$

$C \rightarrow Cd \mid d$

براي تبدیل به LL(1) فرم توسعه یافته ساده تر است. در مورد S عمل فاكتور گیري چپ و در مورد D پس از عمل فاكتور گیري چپ ، باید حالت خود بازگشتي چپ حذف شود. در مورد سایرترم هاي گرامر باید حالت خودبازگشتي چپ از داخل قاعده حذف شود.

$S \rightarrow a\,(B \mid C) \mid dD$

$D \rightarrow d\ \{a \mid b\}$

$B \rightarrow b\ \{C\}$

$C \rightarrow d\ \{d\}$

با جایگزیني گسترش C در $B \rightarrow b\ \{C\}$ این قاعده بصورت زیر تبدیل میشود:

$B \rightarrow b\ \{d\}$

در بین قواعد فوق ، تنها قاعده $S \rightarrow a\,(B \mid C) \mid dD$ ممكن است عاملي براي LL(1) نشدن گرامر باشد. در سمت راست این قاعده 'B | C' را در نظر بگیرید. باید First(B) $\cap$ First(C) $= \varnothing$ باشد.

$$First(B) \cap First(C) = \{b\} \cap \{d\} = \varnothing$$

بنابراین گرامر LL(1) است.

```
Begin   init; NextSymbol;   S(S_EOF);
                                                                End.
(* S  →  a (B | C ) |
                                                                dD *)
Procedure S( Stop : Set of Symbols);


                                                                Begin
 If(CurrentSymbol = S_d)
        Then Begin  NextSymbol; D( Stop );  End
    Else Expect(S_a, [S_b, S_d] + Stop);


                                                                End;
(* D → d  { a  | b
                                                                } *)
    Procedure D( Stop : Set of Symbols);


                                                                Begin
 Expect(S_d, [S_b, S_d] + Stop);
  While(CurrentSymbol = S_a ) Or (CurrentSymbol = S_b) do
   If     CurrentSymbol   =   S_a   Then
                                                                NextSymbol
   Else Expect(S_a, [S_b, S_d] + Stop);
```

```
                                                                    End;
(*B → b { d }*)
    Procedure B( Stop : Set of Symbols);

                                                                    Begin
Expect(S_b,   [   S_d]   +
                                                                    Stop);
 While(CurrentSymbol = S_d ) do NextSymbol;

                                                                    End;
(*B → b { d }*)
    Procedure C( Stop : Set of Symbols);

                                                                    Begin
Expect(S_d,   [   S_d]   +
                                                                    Stop);
 While(CurrentSymbol = S_d ) do NextSymbol;

                                                                    End;
```

٤.٩ مولد تحليلگر نحوى

مولد تحليلگر نحوي برنامه اي است كه گرامر يك زبان را دريافت نموده و براي آن يك تحليلگر نحوي يا تجزيه گر ايجاد مي كند. در اين قسمت نوعي خاص از مولد ارائه شده است كه گرامر زبان را در داخل يك فايل ميپذيرد و بر اساس گرامر عمل تحليل نحوي را انجام ميدهد. اين مولد در واقع يك برنامه پيمايش گراف است. هر قاعده درگرامر LL(1) را ميتوان بعنوان يك گراف در نظر گرفت.

دربرنامه مولد تحليلگر نحوي كه درزير ارائه شده ، گرامر درون يك فايل متن يا در اصطلاح Text بنام Grammar.txt قرار دارد. براي قرار دادن گرامر در داخل اين فايل در هر سطر ابتدا ترم سمت چپ قاعده و بلافاصله گسترش آن مشخص ميشود. براي سادگي و خواناتر شدن برنامه فرض شده كه هر ترم مياني و سرترم با يك حرف لاتين درشت و هر ترم پايانى با يك حرف كوچك مشخص شده است. به اين ترتيب براي نمونه گرامر ساده :

S → cA | BdS
A → aAB |
d
B → bB |
d

بصورت زير در فايل Grammar.txt ذخيره ميشود :

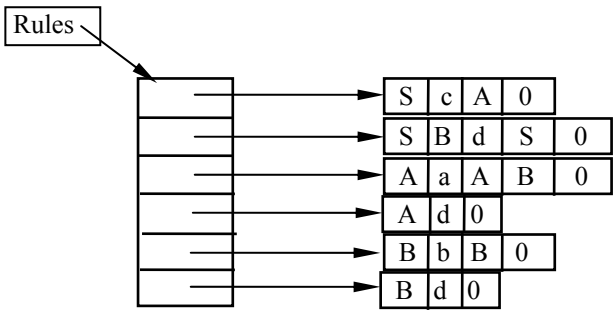
6
ScA
SBdS
AaAB
Ad
BbB
Bd

در اولين سطر از فايل Grammar.txt تعداد قواعد يا در واقع تعداد سطرهاي فايل مشخص شده است. بدنه برنامه اصلي main در زير ارائه شده است. بازهم بخاطر سادگي و خوانايي برنامه متن برنامه مورد كامپايل در داخل يك رشته بنام Statement در نظر گرفته شده است.

```
        void main()
{  int NoRules,        تعداد قواعد در گرامر
                                                    //
   len;                // تعداد ترمها در جمله ورودي
   char **rules,        ماتريس قواعد گرامر
                                                    //
```

```
*statement;   جمله ورودي كامپايلر
                                                                    //
// توليد ماتريس قواعد گرامر در آرايه دو بعدي
                                                                    1- rules
        NoRules = ReadGrammar(&rules, statement);
تحليل نحوي جمله داده شده statement با استفاده از ماتريس قواعد rules
                                                                    // 2-
        len = TopDownParse(rules, statement, 0, NoRules, 0);
//3. اعلام پيام خطا در صورتيكه كار تحليل نحوي تا آخر جمله داده شده ادامه پيدا نكرده
باشد.
    if(len != strlen(statement)) { clrscr(); puts("خطا نحوي"); }
            }
```

در اولين سطر از برنامه اصلي تابع ReadGrammar فراخواني ميشود. اين تابع گرامر را از داخل فايل Grammar.Txt خوانده ، در داخل يك آرايه از نشانگرها به آرايه ها قرار ميدهد. آدرس شروع آرايه نشانگرها در مكان مشخص شده توسط پارامتر Prod از تابع ظاهر ميشود. هر آرايه حاوي يك قاعده گرامر خواهد بود. به عنوان نمونه براي گرامري كه در ابتداي اين بخش ارائه شد ساختار آرايه كه Rules نام دارد به صورت زير است :



كد تابع ReadGrammar در زير ارائه شده است.

```
   int ReadGrammar(char ***prod, char *statement)
گرامر را از فايل ورودي به ماتريس مربوطه انتقال ميدهد . ازانتهاي فايل جمله ورودي
را ميخواند // {
          FILE *fp;
  char line[100], **rules;
         int i, NoRules;
         fp = fopen("Gammar2.txt","rt");
        if(!fp) fp = stdin;
         if(fp == stdin)
       {clrscr(); printf("NO. Rules : ");}
       fscanf(fp,"%d", &NoRules);
ايجاد آرايه از نشانگرها بسول تعداد قواعد گرامر باضافه
                                                                    يك //
    rules =(char **) malloc((NoRules+1) * sizeof(char *))  ;
        rules[NoRules] = NULL;
     for(i = 0; !feof(fp) && i < NoRules; i++)
قرار دادن قواعد درون ماتريس {
                                                                    // Rules
```

```
        fscanf(fp,"%s",line);
    rules[i] = malloc(strlen(line)+1);
        strcpy(rules[i], line);
    }
     خواندن جمله مورد
                                                                    کامپایل //
    if(!feof(fp)) fscanf(fp, "%s", statement) ;
*prod = rules;    // برگرداندن آدرس ماتریس قواعد در
                                                                    Prod
        برگرداندن تعداد قواعد بعنوان خروجي تابع return
                                                                NoRules; //
          }
```

اکنون با تشکلیل ماتریس قواعد درRules و خواندن جمله مورد کامپایل در رشته Statement میتوان با فراخواني تابع TopDownParse عمل تحلیل نحوي را انجام داد. این تابع براي انجام عمل تحلیل نحوي از Rules[start] شروع مي کند. در آغاز کار مقدار start مساوي با صفر است لذا، در آغاز کار تحلیلگر نحوي با سرترم گرامر آغاز میشود. ترم پیش بیني براي این تابع در Statement[ix] قرار گرفته است.

تابع تحلیلگر

نحوي //

```
int TopDownParse(char **rules, char *statement, int start, int NoRules, int ix)
    پارامتر Rules حاوي قواعد گرامر
                                                                    است .// {
      پارامتر start حاوي شماره قاعده ترم مياني مورد
                                                                انتظار است .//
    پارامتر ix حاوي انديس ترم پيش بيني در جمله ورودي
                                                                    است//
         int i, j, k, m, NewK;
        char leftterm, input;
        //1- Match the leftmost
     input = statement[ix];  // تعیین ترم پیش بیني در متغیر
                                                                    Input
با فراخواني تابع StartWhichRules مقدار i نشان میدهدکه در داخل آرایه //
                                                                    Rules
  کدام گسترش از گسترش‌هاي ترم مورد انتظار يعني rules[start] با ترم پيش بيني Input
آغاز میشود.//
      i = StartWhichRule(rules, input, start, NoRules);
          if(i < 0) return ix;
      // پیمایش سمت راست قاعده
                                                                rules[i][0]
    for( j = 1, k = ix; rules[i][j] && statement[k] && i < NoRules;  j++)
// اگر ترم بعدي در سمت راست قاعده يك ترم مياني است آنگاه
            if(isupper(rules[i][j]))
                {
جستجو براي يافتن قاعده مربوط به ترم مياني ظاهر شده در ضمن پيمايش سمت راست
                                                                    // 1-
```

```
        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 بعدي در سمت راست قاعده يك ترم پايانی
                                                                    است //
  else    خواندن ترم پايانی بعدي در ورودي
                                                                    // 2.
          if(rules[i][j] == statement[k]) k++;
        return k;
  }
```

با فراخواني تابع StartWhichRule مشخص ميشود كه ترم پيش بيني input كدام گسترش از گسترش‌هاي ترم مورد انتظار يعني rules[start][0] را آغاز ميكند. در صورت يافتن گسترش مورد نظر شماره انديس قاعده آن در متغير i برگردانده ميشود. به اين ترتيب rules[i][0] بايد مساوي با rules[start][0] باشد. با يافتن يك قاعده مناسب ، سمت راست آن قاعده مورد پيمايش قرار ميگيرد. در ضمن پيمايش چنانچه تحليلگر به يك ترم مياني برسد ، با فراخواني خود بطور خود بازگشتي بر مبناي آن ترم مياني عمل تحليل نحوي را ادامه ميدهد. پس از اين فراخواني NewK نمايانگر انديس ترم پيش بيني درجمله ورودي يعني Statement است. واضح است كه مقدار اين انديس بايد متفاوت از انديس قبل از فراخواني خودبازگشتي تابع يعني K باشد.

تابع StartWhichRule مشخص ميكند كه ترم پيش بيني input آغاز كننده كدام گسترش از گسترش‌هاي متفاوت ترم rules[start][0] است. براي اين منظور چنانچه LeftNonTerm مساوي با rules[start][0] قرار داده شود، تحليلگر بدنبال گسترشی از اين ترم مياني است يا به عبارت ديگر بدنبال يك rules[i] ميگردد كه اولا" rules[i][0] مساوي با LeftNonTerm باشد و ثانياً ترم rules[i][1] يا ترم مورد پيش بيني يعني input آغاز شود. اگر rules[i][1] يك ترم مياني باشد ، تابع بصورت خود بازگشتي فراخواني ميشود تا مشخص شود آيا آن ترم مياني با ترم پيش بيني يعني input آغاز ميشود. اگر موفق نشد به سراغ گسترش ديگر ترم مياني ميرود. در اين مرحله ميتوان با استفاده از مجموعه سرآغاز كار تحليلگر را تسريع نمود. برنامه كامل مولد تحليلگر نحوي در زر ارائه شده است.

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


int ReadGrammar(char ***prod, char *statement)
        {
        FILE *fp;
char line[100], **rules;
```

```
        int i, NoRules;
         fp = fopen("Gammar2.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);
            *prod = rules;
            return NoRules;
    } // End of ReadGrammar

  //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];
     //1- Look for a left terminal in the production rule
         i = start;
      while((rules[i][1] != input) &&
                                  (rules[i][0] == LeftNonTerm)) i++;
 //Hint: here is a catch --- E R O R R -- Find it out..
      if(rules[i][1] == input)
         return i;
    //2- 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 = StartWhichRule(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;

//1- 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 = ReadGrammar(&rules, statement);
     len = TopDownParse(rules, statement, 0, NoRules, 0);
     if(len != strlen(statement)) { clrscr(); puts("ERROE");  }
           }
```

٤.١٠ تمرین

تمرین 1- گرامر زیر براي ساختار لیست را در نظر بگیرید :

$S \rightarrow (L) \mid a$
$L \rightarrow L , S \mid S$

این گرامر را به فرم LL(1) تبدیل نموده براي آن جدول تجزیه بالا به پائین ایجاد کنید. با استفاده از این جدول تجزیه جمله (a, (a, a)) را مورد تحلیل نحوي قرار دهید.

تمرین 2- گرامر زیر را به فرم LL(1) تبدیل نموده جدول تجزیه براي یك تحلیلگر پیش بیني کننده ایجاد نمائید.

$S \rightarrow SAB \mid AB$
$A \rightarrow Aa \mid a$
$B \rightarrow Bb \mid \varepsilon$

تمرین 3- یك الگوریتم براي تحلیل گر پیش بیني کننده ایجاد کنید که با استفاده از جدول تجزیه و یك پشته تجزیه عمل تحلیل نحوي را انجام دهد. براي این الگوریتم یك تابع بصورت زیر ایجاد کنید :

```
PredictiveParser( char **ParseTable,  int  NoRows, int NoCols)
                                                              int
```

تمرین 4- یك مولد تحلیل گر پیش بیني کننده ایجاد کنید که گرامر LL(1) را در ورودي مي پذیرد. این مولد سپس ، مجموعه هاي سرآغاز براي ترمهاي میاني را محاسبه میکند. با استفاده از مجموعه هاي سرآغاز به راحتي میتوان جدول تجزیه را تولید کرد.

تمرین 5- گرامر زیر را به فرم توسعه یافته LL(1) تبدیل نموده ، یك تحلیل گر کاهینه بازگشتي براي آن ایجاد کنید. در ضمن مسإله بهبود از خطا را نیز در نظر بگیرید.

$S \rightarrow SbB \mid SaB \mid L\ a\ A$
$L \rightarrow L\ a\ B \mid LbB \mid \varepsilon$
$A \rightarrow b\ A \mid d$
$B \rightarrow Bb \mid \varepsilon$

تمرین 6- گرامر ارائه شده در تمرین 5 را تبدیل به فرم LL(1) نمائید و سپس جدول تجزیه بالا به پائین براي آن ایجاد کنید.

تمرین 7- گرامر زیر را به فرم LL(1) تبدیل کنید :

$A \rightarrow \alpha$
$A \rightarrow A_1\ \alpha_1$
$A_1 \rightarrow A_2\ \alpha_2$

... ...

... ...

$A_n \rightarrow A\ \alpha_{n+1}$

تمرين 8- يك الگوريتم كلي براي حذف گسترش‌هاي تهي در گرامر زبانها ارائه دهيد.

تمرين 9- گرامر زير را به فرم LL(1) تبديل نموده يك تجزيه گر كاهينه بازگشتي براي آن ايجاد كنيد. مسئله بهبود از خطا را نيز در نظر بگيريد.

$S \rightarrow SAB \mid Bd$

$A \rightarrow BdA \mid dB \mid \varepsilon$

$B \rightarrow Bb \mid \varepsilon$

تمرين 10- چگونه ميتوان براي يك تجزيه گر پيش بيني كننده مسإله بهبود از خطا را در نظر گرفت. الگوريتم خواسته شده در تمرين 3 را با در نظر گرفتن مسإله بهبود از خطا تكميل كنيد.

تمرين 11- برنامه يك تجزيه گر كاهينه بازگشتي براي گرامر يكي از زبانهاي رايج برنامه نويسي ، ايجاد كنيد. مسئله بهبود از خطاء در نظر گرفته شود.

تمرين 12- مولد تحليلگر نحوي ارائه شده در انتهاي فصل را آنچنان تكميل نمائيد كه تحليلگر لغوي را فراخواني كند تا ترم بعدي را از ورودي دريافت كند. ترمهاي مياني نيز به هر صورتي در گرامر ظاهر شوند. با محاسبه اتوماتيك مجموعه First براي ترمهاي مياني و سرترم ميتوان كار مولد تحليلگر را تسريع نمود. مسإله بهبود از خطا نيز در نظر گرفته شود.

تمرين 13- گرامر زير را به فرم LL(1) تبديل نماييد و سپس با در نظر گرفتن مسئله بهبود از خطاء براي آن يك تحليلگر كاهينه بازگشتي ايجاد كنيد.

$S \rightarrow Abd \mid Bd$

$A \rightarrow Aa \mid$

a

$B \rightarrow Bb \mid$

b

تمرين 14- گرامر زير را به فرم LL(1) تبديل نموده ، يك تحليلگر كاهينه بازگشتي براي آن ايجاد كنيد.

$S \rightarrow AB \mid A\ B\ C$

$A \rightarrow A\ a \mid$

ε

$B \rightarrow b\ A \mid A\ b\ b$

$C \rightarrow C\ c \mid \varepsilon$

تمرین 15- در حالت کلي چگونه میتوان اثبات کرد که یك گرامر به فرم LL(1) قابل تبدیل است.

فصل پنجم

تجزیه پایین به بالا

۵.۱ مقدمه

در روش تجزیه پائین به بالا مراحل تجزیه از ترمهاي پایانی درون جمله داده شده آغاز، وبه سر ترم گرامر خاتمه مي یابد . تجزیه گرهاي پائین به بالا گرامرهاي گسترده تري نسبت به تجزیه گرهاي بالا به پائین را مي پوشانند. در این فصل انواع روشهاي اتوماتیك تجزیه پائین به بالا مطرح خواهد شد و چگونگی ترسیم جداول تجزیه ، به روشهایي موسوم به **LR , LALR , SLR** مطرح میگردد . نکته قابل توجه استفاده از گرامرهاي مبهم است که موضوع آخر این فصل را به خود اختصاص میدهد .

روش LR(k) براي تجزیه پائین به بالا با استفاده از K ترم بعدي در جمله مورد کامپایل در هر مرحله از تجزیه استفاده مي نماید. چون عمل خواندن لغات از داخل جمله ورودي مورد کامپایل از چپ به راست انجام میشود حرف L که مخفف Left to Right مي باشد، در نام این روش ظاهر شده است. حرف R مخفف Rightmost derivation یا استنتاج راست است. کلمه LALR مخفف Lookahead LR است. روش LALR(k) نیز براي دسته اي محدودتر از گرامرها نسبت به روش LR(k) مورد استفاده قرار میگیرد. کلمه SLR مخفف Simple LR است. در این فصل روشهاي LR(1) ، LALR(1) و SLR(1) مورد بررسي قرار خواهد گرفت.

۵.۲ اصول تجزيه پايين به بالا

اصولاً در روش تجزيه بالا به پائين پس از اينكه تحليلگر نحوي لغات را دريافت نمود دو عمل را ممكن است انجام دهد. تحليلگر يا لغت را مستقيما" به داخل يك پشته بنام پشته تجزيه انتقال ميدهد و يا اينكه بر اساس لغت در يافت شده ترمهاي بالاي پشته كه با سمت راست يك قاعده از قواعد گرامر مطابقت دارند را از بالاي پشته خارج نموده ، با ترم مياني سمت راست قاعده جايگزين ميكند. با جايگزيني ترمهاي بالاي پشته با ترم مياني سمت چپ قاعده مربوطه در واقع چند ترم بالاي پشته به يك ترم كاهش يافته اند. لذا، اين عمل را در اصطلاح عمل كاهش يا Reduce گويند.عمل انتقال لغات دريافتي از تحليلگر لغوي به داخل پشته تجزيه را در اصطلاح عمل انتقال يا Shift گويند.

عمل كاهش يا Reduce بر اساس يك قاعده از گرامر انجام مي گيرد. براي نمونه ترمهاي بالاي پشته تجزيه كه عيناً مشابه ترمهاي سمت راست قاعده شماره مثلا" n هستند ، به ترم سمت چپ قاعده كه يك ترم مياني يا سرترم گرامر است كاهش داده مي شود. اين عمل را بطور خلاصه با دستور Rn مشخص ميكنند. دستورالعمل Rn مشخص ميكند كه عمل Reduce بر اساس قاعده شماره n انجام ميشود.

اصولا در روش تجزيه پائين به بالا عمل خواندن لغات مثل قبل از چپ به راست جمله داده شده انجام مي شود. ترمها براساس قواعد زبان ، دسته بندي وبه يك ترم مياني در سمت راست قواعد كاهش مي يابند و يا در اصطلاح خارجي Reduce داده ميشوند. يك جمله يا برنامه در داخل يك فايل متن يا در اصطلاح Text قرار ميگيرد. در انتهاي هر فايل متن علامت خاتمه فايل ظاهر ميشود. هر جمله داده شده نهايتا" در صورت صحت از لحاظ فرم گرامري به سرترم گرامر كاهش داده ميشود. بنابراين پس از سرترم گرامر همواره انتظار مشاهده علامت خاتمه فايل مي رود.

علامت خاتمه فايل در اينجا با كاراكتر $ مشخص مي شود . جهت كسب اطمينان از ظهورعلامت خاتمه فايل پس از سرترم گرامر چنانچه براي نمونه سرترم گرامر S باشد ، قاعده

$$S' \rightarrow S\ \$$$

را به ابتداي گرامر افزوده ، در اصطلاح گرامر را به فرم توسعه يافته يا Extended تبديل مي نمايند. بنابراين به ابتداي گرامر عبارات چهار عمل اصلي با سر ترم E ، قاعده :

$$E' \rightarrow E\ \$$$

افزوده شده ، در اصطلاح گرامر عبارات به فرم توسعه يافته تبديل مي شود.

0 $E' \rightarrow E\ \$$

1 $E \rightarrow E + T$

2 $E \rightarrow E - T$

3 $E \rightarrow T$
4 $T \rightarrow T * F$
5 $T \rightarrow T / F$
6 $T \rightarrow F$
7 $F \rightarrow Id$
8 $F \rightarrow No$
9 $F \rightarrow (E)$

بنا بر گرامر فوق ، صحت عبارت : (a - b) * c / d سنجيده ميشود . براي اين منظور از يك پشته تجزيه ، جهت حفظ فرمهاي جمله اي استفاده مي شود .

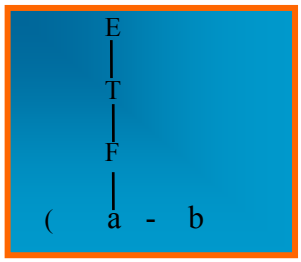
عمليات	ورودی	پشته تجزيه
s,s	(a-b)*c/d $	(A
R7,R6,R3	-b)*c/d $	(A
S,S	)*c/d $	(A
R7	)*c/d $	(A
R6	)*c/d $	(E-T
R2	)*c/d $	(E
S	)*c/d $	(E)
R9	)*c/d $	F
R6	/d $	T
S,S	/d $	T*c
R7	/d $	T*F
R4	$	T
S,S	$	T/d
R7	$	T/F
R5	$	T
R3,S,R0		E

شكل 1و4- مراحل تجزيه جمله (a - b) * c / d

در جدول فوق ، نكته قابل توجه انتخاب عمل كاهش و يا انتقال در هر مرحله از تجزيه بالا به پائين است. تصميم گيري در مورد انتخاب عمل كاهش (Reduce) و يا انجام عمل انتقال (Shift) ، به موقعيت كنوني پشته و فرم جمله اي موجود در آن و بالاخره چگونگی ترم موجود بر سر ورودي كه در اصطلاح ترم پيش بيني يا Look ahead ناميده مي شود ، وابسته است . براي نمونه در شروع عمل تجزيه ابتدا '(' و سپس 'a' از ورودي خوانده مي شود. چون ترم بعدي موجود بر سر ورودي اكنون علامت منها ميباشد و طبق گرامر و بنابر قاعده :

2 $E \rightarrow E - T$

حتماً قبل از منها يك E بايد وجود داشته باشد، تصميم به كاهش a بر اساس قاعده شماره هفت ويا بطور مختصر تصميم به انجام دستورالعمل R7 و سپس R6 و نهايتاً R3 گرفته شد. لذا، تا اين مرحله درخت تجزيه بصورت زير ايجاد ميشود.



در مرحله بعدي با انجام عمل S يا در واقع عمل انتقال لغت '-' از سر ورودي خوانده ميشود. اكنون طبق گرامر پس از علامت '-' بايد در ورودي T ظاهر شود. لذا، ترم بعدي يعني 'b' از ورودي به داخل پشته تجزيه انتقال يا در اصطلاح Shift داده ميشود.

حالا، بر سر ورودي ترم '(' قرار گرفته است . قبل از '(' بر طبق قاعده :

$F \rightarrow (E)$ 9

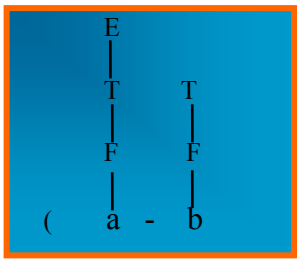
بايد E ظاهر شود. تا اين مرحله طبق جدول تجزيه بر سر پشته فرم جمله أي

'(E - b'

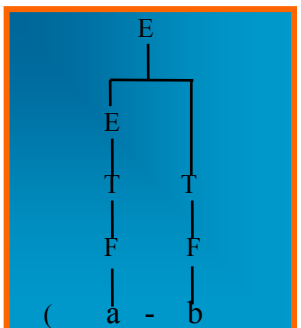
قراردارد و ترم پيش بيني '(' است. لذا، جهت رسيدن به E قبل از '(' عمل R7 و سپس عمل R6 انجام ميشود. بنابراين فرم جمله اي موجود در پشته تجزيه بصورت

'(E - T'

تبديل ميشود. تا اين مرحله از تجزيه پائين به بالا درخت تجزيه بصورت زير است. فرمهاي جمله اي ايجاد شده نيز در داخل پشته تجزيه در بالا مشخص مي باشند.



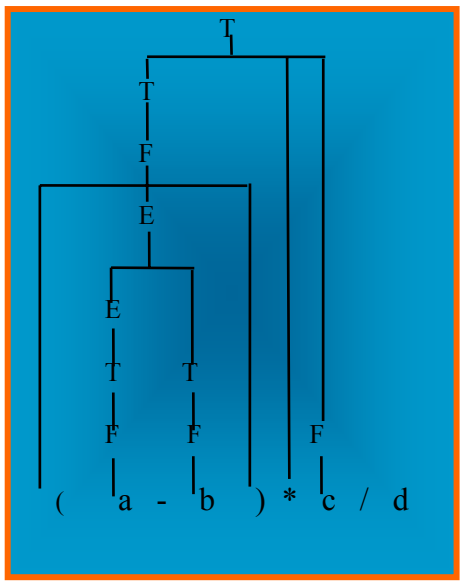
حالا ، ترم پيش بيني '(' و بر سر پشته تجزيه ترم T موجود است. اما بلافاصله T با E جايگزين نشده ، دستورالعمل R3 اجراء نمي شود . بايد توجه نمود كه همواره تنها ترم ثيش بيني عامل تصميم گيرنده نيست بلكه ، چگونگی فرم جمله اي دربالاي پشته تجزيه نيز در تصميم گيري و انتخاب نوع عمل كاهش و يا انتخاب عمل انتقال موثر است . لذا، در اين مرحله عمل R2 انجام شده ، بر سر پشته تجزيه E - T به E بر طبق قاعده شماره 2 كاهش داده ميشود. تا اين مرحله از تجزيه پائين به بالا درخت تجزيه بصورت زير است :



در مرحله بعدي ')' با انجام عمل S به داخل پشته تجزیه انتقال داده میشود. اکنون بر سر ورودي لغت '*' قرار دارد. لذا ، چون قبل از ضرب طبق قاعده

4 $T \rightarrow T * F$

باید T وجود داشته باشد عملیات R9 و سپس R6 انجام میشود. حالا با انجام دو عمل S متوالي ترم هاي '*' و C به داخل پشته تجزیه انتقال داده میشوند. اکنون بر سر پشته تجزیه فرم جمله اي 'T * C' و ترم ثیش بیني '/' است . لذا ، عملیات R7 ، R6 و R4 متوالیاً انجام میشوند. سپس با انجام دو عمل S لغت '/' و 'd' به داخل پشته انتقال داده میشوند. تا این مرحله درخت تجزیه بصورت ریر است.



به همین ترتیب تا انتهاي جمله داده شده عمل تجزیه ادامه دارد.

۵.۳ طرح روشی برای ایجاد جدول تجزیه LR (۱)

در این بخش چگونگی ایجاد الگوریتم تجزیه پائین به بالا استدلال میشود. الگوریتم حاصل بطور خلاصه در انتها ارائه شده است. الگوریتم حاصل تجزیه LR(1)[1]

[1] همانگونه که در مقدمه این فصل توضیح داده شد ، در کلمه LR(1) حرف L مخفف Left to Right و R مخفف Rightmost derivation است.

ناميده مي شود. اين الگوريتم مبتني بر تحليل روش تجزيه كه در بالا توضيح داده شد ايجاد ميشود.

5.3.1 توليد الگوريتم

در بخش قبل مشاهده كرديد كه در حالت كلي براي انجام عمل تجزيه نياز به استفاده از يك پشته تجزيه مي باشد. دو عمل صورت ميگيرد يكي انتقال وديگري كاهش . عمل انتقال و كاهش بر اساس چگونگی ترم پيش بيني بر سر ورودي و وضعيت كنوني پشته تجزيه مشخص مي شد . در بعضي از تجزيه گرهاي LR نياز به بيش از يك ترم پيش بيني براي عمل تجزيه است ، براي نمونه اگر حداكثر K ترم پيش بيني نياز باشد ، تجزيه گر را LR(k) مي نامند .

اكنون با يك استدلال ساده نشان داده خواهد شدكه چگونه ميتوان بصورت اتوماتيك بر اساس جدول تجزيه كه در مورد آن بحث خواهد شد عمل تجزيه پائين به بالا را انجام داد. براي نمونه گرامر زير را در نظر بگيريد :

0 S' → S$
1 S → aBb
2 S → AB
3 A → bA
4 A → b
5 B → Bb
6 B → a

بر طبق گرامر فوق نهايتاً در بالاي درخت تجزيه بايد سرترم S' ظاهر شود. بنابراين آخرين مرحله در تجزيه بالا به پائين كاهش S$ به S' است. اما قبل از اينكه بتوان اين عمل را انجام داد در ورودي ابتدا بايد ترم S ظاهر شود. بنابراين ميتوان گفت در تجزيه پائين به بالا ، در آغاز كار تحليلگر نحوي در انتظار كاهش ورودي يا به عبارت ديگر برنامه يا جمله مورد كامپايل به S است. لذا، ميتوان اين وضعيت انتظار را بصورت زير براي تحليلگر مشخص نمود.

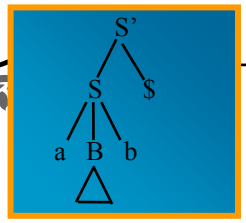
S' → • S$

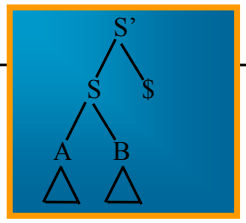
در اين وضعيت ،•، نشان ميدهد كه در انتظار S بايد بود. اما براي تشكيل S بايد طبق گرامر يا عمل R1 يا عمل R2 انجام شود. يعني طبق گرامر

1 S →
aBb

2 S →
AB

رشته aBb يا رشته AB در ورودي ظاهر شود. به عبارت ديگر قبل از خاتمه عمل تجزيه بايد يكي از درختهاي زير ايجاد شوند.





پس از مشاهده سرترم S طبق وضعیت S' → • S $ انتظار میرود که علامت خاتمه فایل $ در ورودي ظاهر شود. پس از مشاهده $ است که میتوان عمل R0 را انجام داد. بنابراین تا این مرحله میتوان گفت که وضعیت در شروع عمل تجزیه بصورت زیر مي تواند باشد.

S' → • S $
S → • a B b , $
S → • A B , $

در بالا علامت $ توسط ویرگول از دو گسترش متفاوت S جدا شده است. به این ترتیب مشخص شده که پس از کاهش سمت راست این دو قاعده به S انتظار میرود که در ورودي $ ظاهر شود. یا به عبارت دیگر انتظار میرود که پس از تشخیص S ترم پیش بیني $ باشد. این موضوع را وضعیت S' → • S $ مشخص نمود.

بنابراین تا کنون مشخص شده است که در حالت شروع و قبل از اینکه هیچ لغتي ازجمله مورد کامپایل در ورودي ظاهر شود ، تحلیلگر نحوي انتظار S' و لذا جهت تشکیل S' در آغاز انتظار کاهش ورودي به S را دارد. پس از مشاهده سرترم S انتظار میرود که علامت خاتمه فایل $ در ورودي ظاهر شود.

براي مشاهده S در آغاز کار انتظار مشاهده a و یا تشکیل ترم میاني A میرود. طبق وضعیت :

S → • a B b , $

ابتدا a سپس B و پس از آن b باید در ورودي ظاهر شوند تا بتوان عمل R1 را انجام داد و S را ایجاد کرد. البته عمل R1 در صورتي موثر است که در سر ورودي $ وجود داشته باشد. به همین ترتیب طبق وضعیت S' → • A B , $ باید ابتدا A ایجاد شود پس از آن B تا بتوان عمل R2 را انجام داد. بنابراین باید ابتدا در ورودي A تشخیص داده شود و پس از تشخیص A باید حتماً در سر ورودي ترمي متعلق به مجموعه First(B) ظاهر شود تا بتوان مطمئن شد که میتوان B را پس از A دید.

براي ایجاد A نیز طبق گرامر باید یا عمل R3 و یا R4 انجام شود. و بعد از انجام عمل حالا بر طبق انتظاري که در بالاتر طبق وضعیت S' → • A B , $ مي رفت ، تحلیلگر نحوي باید یك ترم متعلق به First(B) را در ورودي مشاهده کند تا بتواند به کار خود ادامه دهد و B را در ورودي تشخیص دهد. بنابراین جهت رسیدن به A یکي از دو وضعیت زیر در آغاز کار و قبل از دریافت هر گونه لغتي از تحلیلگر لغوي پیش مي آید.

A → • b A , First(B)
A → • b , First(B)

بنابراين در حالت شروع و قبل از دريافت اولين لغت از تحليلگر لغوي وضعيت هاي مورد انتظار تحليلگر نحوي بصورت زير ميباشد.

I0 : S' → • S $
S → • a B b ,$
S → • A B ,$
A → • b A ,First(B)
A → • b ,First(B)

حالا فرض كنيد كه اولين ترم پايانی يا اولين لغت از تحليلگر لغوي دريافت شود. چه اتفاقي ميتواند رُخ دهد ؟ طبق انتظار تحليلگر كه در بالا مشخص شده است بايد اين لغت b يا a باشد. طبق وضعيت :

'S → • a B b ,$'

انتظار مشاهده a و طبق وضعيتهاي :

'A → • b A ,First(B)'
'A → • b ,First(B)'

در آغاز انتظار مشاهده b ميرود. اگر در ورودي b ظاهر شود حالت جديد زير حاصل ميشود:

I1 : A → b • A ,First(B)
A → b • ,First(B)

حالا اگر در ورودي First(A) ظاهر شود به سراغ گسترش A در ادامه وضعيت b • A بايد رفت و اگر First(B) در ورودي ظاهر شود بايد طبق قاعده A → b عمل كاهش b به A را انجام داد. بنابر 'A → b • A ,First(B)' براي مشاهده A درحالت I1 انتظار ديدن A دومي هم در سمت راست وضعيت ميرود. اما براي مشاهده A در ورودي طبق قواعد 3 و 4 گرامر بايد b A يا b در ورودي ظاهر شود. پس از اين A دوم First(B) بايد درورودي ظاهر شود. بنابراين حالت I1 بصورت زير تبديل ميشود:

I1 : A → b • A ,First(B)
A → b • ,First(B)
A → • b A ,First(B)
A → • b ,First(B)

در حالت I1 طبق وضعيت 'A → b • A ,First(B)' انتظار مشاهده A پس از b در سمت راست وضعيت ميرود. پس از A انتظار مشاهده First(B) ميرود . بنابراين ، براي تشكيل A در ورودي دو وضعيت :

A → • b A ,First(B)
A → • b ,First(B)

به حالت I1 افزوده شده است. به اين ترتيب در اين حالت پس از اينكه طبق يكي از دو وضعيت فوق در ورودي A تشخيص داده شد وضعيت 'A → b • A ,First(B)' كه در انتظار تشكيل A در ورودي است تغيير حالت داده جالت جديد I5 تشكيل ميشود.

I5: A → b A • First(B)

حالا مي توان عمل R3 را انجا داد. در حالت I1 طبق وضعيت 'A → b • ,First(B)' مشخص شده كه در ورودي b ظاهر شده است. اكنون ، در صورتي كه ترم پيش بيني عنصري متعلق به First(B) باشد ، مي توان اذعان داشت كه در ورودي A مشاهده شده است. بنابراين ميتوان به حالت I0 برگشت و اعلام كرد كه برطبق انتظار A در ورودي مشاهده شد.

اكنون ، درحالت I0 وضعيت 'S → • A B ,$' كه در انتظار مشاهده A در ورودي بوده به وضعيت جديد:

'S → A • B ,$'

تبديل ميشود. اين وضعيت هسته حالت جديد تري بنام I2 بصورت زير ميشود.

I2 : S → A • B ,$
B → • B b ,$, b
B → • a ,$, b

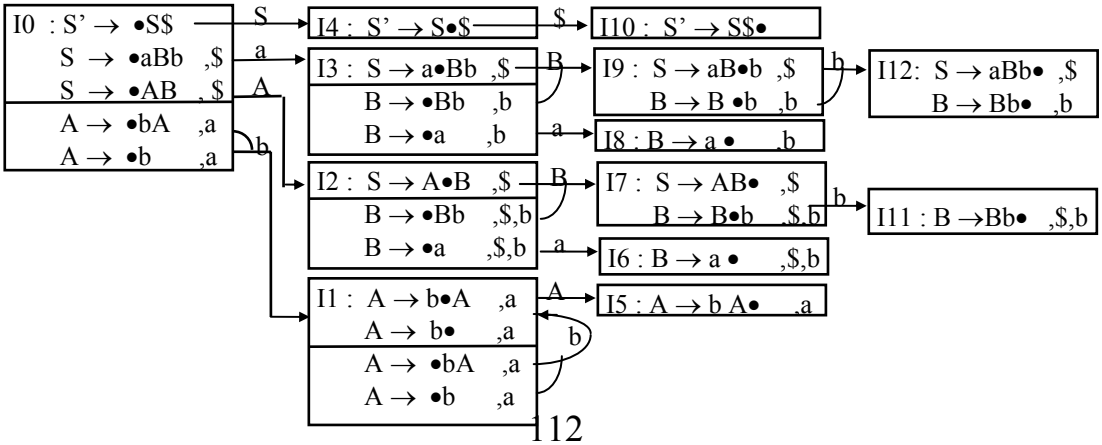
در هسته حالت I2 طبق وضعيت 'S → A • B ,$' انتظار مشاهده B ميرود پس از B نيز همانند S انتظار مشاهده $ در اين وضعيت ميرود. بنابراين بايد گسترشهاي مختلف B با ترم پيش بيني $ را ايجاد كرد. بنابراين دو وضعيت زير به اين حالت افزوده ميشود.

B → • Bb ,$
B → • a ,$

حالا ، طبق وضعيت 'B → • Bb ,$' انتظار مشاهده B در سمت راست ميرود و پس از B انتظار ديدن b در ورودي ميرود. براي مشاهده B طبق گرامر بايد Bb يا a در ورودي ظاهر شود. بنابراين بايد دو وضعيت 'B → • Bb ,b' و 'B → • a ,b' به اين حالت افزوده شوند. امآ، چون قبلا" اين دو وضعيت با ترم پيش بيني $ وجود دارند مي توان ترم پيش بيني a را فقط به آن افزود. بنابراين دو وضعيت زير به حالت I2 افزوده شده است.

B → • B b ,$, b
B → • a ,$, b

اگر به اين ترتيب ادامه داده شود ، گراف تجزيه LR(1) بصورت زير ايجاد ميشود:



شكل 5.1- گراف تجزيه LR(1)

5.3.2 الگوريتم توليد گراف تجزيه **LR(1)**

بطور خلاصه براي توليد گراف حالات كه يك نمونه از آن در شكل 5.1 نمايش داده شد ، بايد مراحل زير انجام شود :

1- عمل توليد گراف از ايجاد حالت شروع I_0 آغاز ميشود. چنانچه سر ترم گرامر S باشد، در هسته يا در اسظلاح Kernel اين حالت وضعيت

$$I0 : \quad S' \rightarrow \bullet\ S\ \$$$

گنجانده ميشود. علامت • قبل از S مشخص ميكند كه در آغاز انتظار مشاهده S ميرود.

2- براي هر وضعيت

$$A \rightarrow \alpha \bullet B\ \beta\ ,\ \delta$$

از يك حالت كه در آن α و يا β هر رشته اي از ترمها و يا اينكه تهي مي تواند باشد ، بايد كليه گسترشهاي ترم مياني B بصورت وضعيتهاي :

$$B \rightarrow \bullet\ \eta \quad , First(\beta)$$

را به آن حالت افزود. ترم پيش بيني براي اين گسترشها (First(β است و چنانچه β وجود نداشته باشد ، آنگاه ترم پيش بيني δ است. براي نمونه يكي از وضعيت هاي حالت I_0 در گراف شكل 5.1 وضعيت $S' \rightarrow \bullet AB \ , \$$ است. چون پس از • ترم مياني A قرار دارد ، گسترشهاي مختلف A با ترم پيش بيني FIrst(B) كه مساوي a است، در نظر گرفته شده است.

$$A \rightarrow \bullet bA \ ,a$$
$$A \rightarrow \bullet b \ ,a$$

3- ترم پيش بيني براي وضعيت هاي خودبازگشتي با فرم كلي

$$A \rightarrow \bullet A\alpha\ ,\ \delta$$
$$A \rightarrow \bullet \beta,\ \ \delta$$

بصورت

$$A \rightarrow \bullet A\alpha\ ,\ \delta, First(\alpha)$$
$$A \rightarrow \bullet \beta,\ \ \delta, First(\alpha)$$

است. براي نمونه در حالت I_2 از گراف شكل 5.1 براي وضعيت : $S' \rightarrow A \bullet B \ ,\$$ طبق معمول دوگسترش ترم مياني B با ترم پيش بيني $ ايجاد ميشود :

$$B \rightarrow \bullet Bb \ ,\$$
$$B \rightarrow \bullet a \ ,\$$

اكنون پس از• ترم مياني B وجود دارد لذا، بايد وضعيتهاي :

$$B \rightarrow \bullet Bb \ ,b$$
$$B \rightarrow \bullet a \ ,b$$

را به حالت I2 افزود. بنابراین با ترکیب این دو وضعیت با وضعیتهاي بالا دو وضعیت زیر حاصل میشود:

$B \rightarrow \bullet Bb$,b, $
$B \rightarrow \bullet a$,b, $

5.3.3 خلاصه عملیات در گراف تجزیه

اگر به گراف شکل 5.1 توجه نمایید. سه نوع عملیات در آن مستتر است.

1-عمل کاهش یا Reduce : عمل کاهش در بالا توضیح داده شد. براي نمونه در حالت I11 طبق وضعیت 'B $\rightarrow$ B a $\bullet$,$, a' چنانچه بر سر ورودي یکي از دو ترم پایانی $ و یا a ظاهر شود میتوان بر طبق قاعده 'B $\rightarrow$ B a' عمل کاهش را انجام داد یا به عبارت دیگر میتوان طبق قاعده شماره 5 عمل R5 را انجام داد. در حالت کلي عمل Rn به مفهوم کاهش بر اساس قاعده شماره n است.

2-عمل *انتقال* یا Shift : عمل دیگر که در گراف مستتر است ، عمل انتقال است. براي نمونه در حالت I0 با مشاهده b در ورودي ، b به داخل پشته تجزیه انتقال داد میشود و به حالت I1 گذر میشود. این عمل را بطور خلاصه با S1 نمایش میدهند. در حالت کلي عمل Sn به مفهوم انتقال و سپس گذر به حالت $_n$است.

3-عمل GoTo : عمل سوم عمل رفتن مستقیم یا در اصطلاح GoTo از یك حالت به حالت دیگر است. براي نمونه در حالت I0 با تشکیل A در سر پشته گذري به حالت 2 بلافاصله انجام میشود. بطور خلاصه عمل Goto 2 انجام میشود. در حالت کلي با تشکیل یك ترم میاني مورد انتظار بر روي پشته میتوان عمل GoTo را به حالت جدیدتر انجام داد.

5.3.4 ایجاد جدول تجزیه

جدول تجزیه در واقع نمایش ماتریسي گراف تجزیه است. عملیات انتقال و GoTo موجب تغییر حالت و یا گذر بین گره هاي گراف میشود. براي نمونه جدول تجزیه شکل 5.1 را میتوان به جدول زیر تبدیل نمود.

0 S' $\rightarrow$ S$
1 S $\rightarrow$ aBb
2 S $\rightarrow$ AB
3 A $\rightarrow$ bA
4 A $\rightarrow$ b

	a	b	$	S	A	B
0	S3	S1	-	4	2	-
1	-	S1	-	-	5	-
2	S6	-	-	-	-	7
3	S8	-	-	-	-	9
4	-	-	S10	-	-	-
5	R3	-	-	-	-	-
6		R6	R6			

5 B → Bb
6 B → a

شكل 5.2- جدول تجزيه LR(1)

اكنون با استفاده از جدول فوق مي توان به راحتي عمل تجزيه بالا به پائين را انجام داد. براي نمونه عبارت abab را توسط جدول فوق بصورت زير مي توان مورد تجزيه پائين به بالا قرار داد.

پشته تجزيه	ورودي	دستور العمل
0	aabbb$	S3
0 a 3	abbb$	S8
0 a 3 a 8	bbb$	R6
0 a 3 B	bbb$	GoTo 9
0 a 3 B 9	bbb$	S12
0 a 3 B 8 b 12	bb$	R5
0 a 3 B	bb$	GoTo 9
0 a 3 B 9	bb$	S12
0 a 3 B 8 b 12	b$	R5
0 a 3 B	b$	GoTo 9
0 a 3 B 9	b$	S12
0 a 3 B 8 b 12	$	R2
0 S	$	S10
پذيرش		

مثال - براي مشخص شدن ... LR (1) ... زير توجه نمائيد :

S → C C
C → a C
C→ d

گرامر ساده فوق را در نظر بگيريد ، هدف ايجاد جدول تجزيه LR (1) براي اين گرامر است. همانگونه كه قبلاً نيز تذكر داده شد ، ابتدا مي بايست گرامر را به فرم توسعه يافته تبديل نمود و پس از آن قواعد را شماره گذاري نمود . به اين ترتيب اين گرامر بصورت زير تبديل ميشود.

0 S’ → S $
1 S → C C
2 C → a C
3 C → d

اكنون ميتوان در مورد حالات ممكن تجزيه شروع به استدلال نمود. به اين ترتيب كه در حالت شروع وضعيت S’ → • S $ است . در اينجا علامت '•' پس از فلش

، چگونگی وضعیت را مشخص میکند . هدف تشخیص S ودر نهایت مشاهده علامت $ است.

پس باید در حالت شروع ، در ورودي يك S را مشاهده کرد یا انتظار دیدن يك S مي رود و پس از آن انتظار میرود که در ورودي $ ظاهر شود. اما طبق گرامر براي مشاهده S باید در ورودي C C را دید، لذا وضعیت در حالت شروع به صورت زیر تبدیل مي شود :

$S' \rightarrow \bullet S \$$
$S \rightarrow \bullet C C$

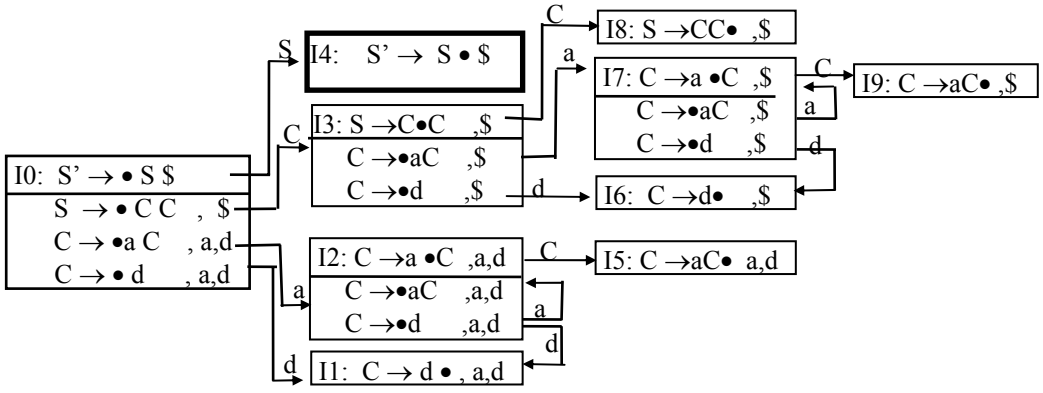
طبق وضعیت $S' \rightarrow \bullet S \$$ ، پس از S انتظار دیدن $ میرود. پس ، پیش بیني میشود که بعد از Cنیز در ورودي $ ظاهر شود . حالا در این وضعیت انتظار میرود که در اولین مرحله در ورودي يك C ظاهر شود و بعد از این C اول C دومي ظاهر شود :

$S' \rightarrow \bullet S \$$
$S \rightarrow \bullet C C \quad , \$$
$C \rightarrow \bullet a C \quad , First(C)$
$C \rightarrow \bullet d \quad , First(C)$

همانگونه که مشاهده مي شود طبق وضعیت $S \rightarrow \bullet C C \quad , \$$، انتظار مشاهده يك C در ورودي میرود. لذا، باید C در ورودي تشخیص داده شود. براي تشخیص C باید طبق گرامر یا aC و یا ترم پایانی d طبق گرامر در ورودي ظاهر شود. در حالت شروع اگر در ورودي d ظاهر شود حالت جدید I1 تولید مي شود. حالت I1 تنها شامل يك وضعیت :

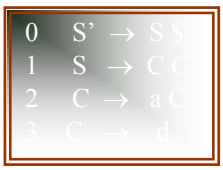
$C \rightarrow \bullet d \quad , First(C)$

خواهد بود. پس از مشاهده d در ورودي ، میتوان به حالت I0 برگشت و اعلام نمود که در وضعیت $S \rightarrow \bullet C C \quad , \$$، انتظار مشاهده يك C میرفت و حالا در ورودي C تشخیص داده شده است. به این ترتیب حالت جدید I3 با وضعیت $S \rightarrow C \bullet C \quad , \$$، در هسته آن ظاهر میشود. پس اینجا تلاش براي دیدن C دوم در ورودي و مشاهده $ بعد از C دوم آغاز میشود. بطور کلي گراف تجزیه بر طبق استدلال فوق به صورت زیر ایجاد میشود :



شکل 5.5 - گراف تجزیه LR (1)

جدول تجزیه براي گراف فوق بصورت زیر است.



	Action			GoTo	
	a	d	$	S	C
0	S2	S1		4	3
1	R3	R3			
2	S2	S1			5
3	S7	S6			8
4	Accept				
5	R2	R2			
6			R3		
7	S7	S6			9
8			R1		
9			R2		

شکل 5.6- جدول تجزیه LR(1)

مثال : جدول تجزیه LR(1) براي گرامر زیر ایجاد کنید :

0 S' → S$
1 S → SA
2 S → AB
3 B → BA
4 B → b
5 A → Aa
6 A → a

گرامر فوق در قرم توسعه یافته است. پس مي توان گراف تجزیه رابلافاصله ایجاد کرد. چون تعداد حالات نسبتا" زیاد است ، هر یك از حالات را بصورت جداگانه مشخص نموده سپس ، با استفاده از یك گراف ارتباط بین حالات ترسیم میشود.

I0 : S' → •S$
S → • SA ,$,a
S → • AB ,$,a
A → • Aa ,b,a
A → •a ,b,a

I1 : A → a • ,b,a

I2: S → A • B ,$,a
A → A • a ,b,a
B → • BA ,$,a
B → • b ,$,a

I3: S' → S • $
S → S • A ,$,a
A → • Aa ,$,a
A → •a ,$,a

I4: B → b • ,$,a

I5: S → AB • ,$,a
B → B•A ,$,a
A → • Aa ,$,a
A → •a ,$,a

I6: A → Aa• ,b,a

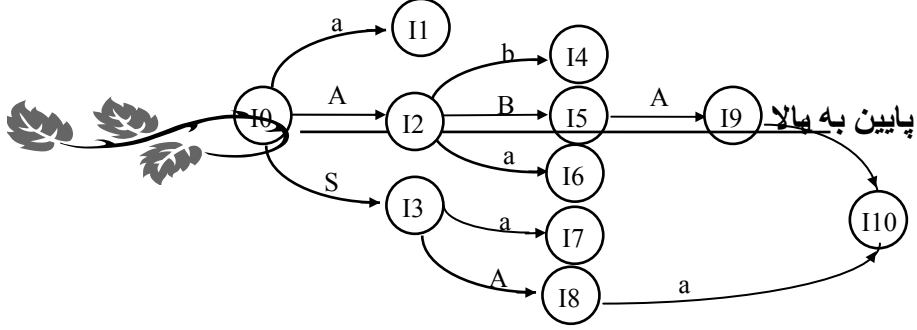
I7: A → a • ,$,a

I8: S → SA• ,$,a
A → A•a ,$,a

I9: B → BA• ,$,a
A → A • a ,$,a

I10: A → Aa• ,$,a

گراف تجزیه بصورت زیر است :



جدول تجزیه LR(1) وگرامر در زیر ارائه شده است.

	a	b	$	S	A	B
0	S1			13	12	
1	R6	R6				
2	S6	S4				5
3	S7		ثذیر		8	
4	R4		R4			
5	R2		R2		9	
6	R5		R5			
7	R6		R6			
8	R1,S10		R1			
9	R3,S10		R3			
10	R5		R5			

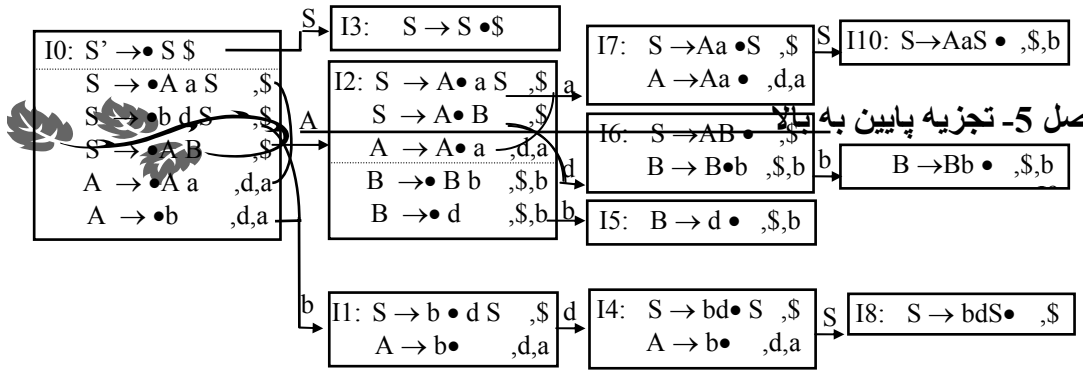
همانگونه که مشاهده مي کنید ، در دو حالت I8 و I9 در داخل جدول تجزیه مشخص نیست که با مشاهده a آیا باید عمل کاهش و یا عمل انتقال را انجام داد. در اصطلاح ، در این دو حالت اختلال در کاهش و انتقال[2] وجود دارد لذا، گرامر LR(1) نیست.

مثال - جدول تجزیه LR(1) براي گرامر زیر ایجاد کنید.

0 $S' \rightarrow S\ \$$
1 $S \rightarrow A\ a\ S$
2 $S \rightarrow b\ d\ S$
3 $S \rightarrow A\ B$
4 $A \rightarrow A\ a$
5 $A \rightarrow b$
6 $B \rightarrow B\ b$
7 $B \rightarrow d$

گراف تجزیه LR(1) براي گرامر فوق بصورت زیر است :

[2] اختلال در کاهش و انتقال را در اصطلاح Shift Reduce Conflict نیز مي گویند. این مشکل موجب میشود که با در دست داشتن یك ترم ثیش بیني نتوان تصمیم گرفت عمل Shift باید انجام شود یا Reduce. ممکن است Reduce Reduce Conflict نیز در برخي موارد ایجاد شود.



گرامر فوق LR(1) نیست زیرا ، در حالت I1 با مشاهده d در ورودي مشخص نیست که آیا عمل R5 و یا عمل S4 باید انجام شود. بعبارت دیگر نمي توان مشخص کرد که آیا عمل انتقال باید انجام شود و یا عمل کاهش. بنابراین ، در این حالت اختلال در کاهش و انتقال و یا در اصطلاح Shift Reduce Conflict وجود دارد.

۵.٤ مشکل گرامرهای LR (۱)

اگر توجه نموده باشید در گرامر **LR(1)** جدول تجزیه نسبت به تعداد قواعد گرامر بسیار بزرگ مي باشد. براي نمونه براي گرامر کوچک :

$S \rightarrow CC$
$C \rightarrow a\ C$
$C \rightarrow d$

جدول تجزیه داراي ده ردیف و پنج ستون بود . به این ترتیب جدول تجزیه براي یك گرامر با فقط سه قاعده تعداد پنجاه خانه داشت. براي گرامرهاي واقعي با حدود صد قاعده ، جدول تجزیه حداقل ده هزار خانه از حافظه را اشغال مي کند. جهت رفع این مشکل سعي شده که گرامر را محدود کنند وگرامرها با امکانات کمتر از گرامرهاي LR (1) جهت بیان قواعد استفاده شود و یا اینکه گرامرها به صورت مبهم استفاده شوند. براي مثال گرامر عبارات بصورت :

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T / F \mid F$
$F \rightarrow Id \mid No \mid (E)$

را مي توان بصورت مبهم و با یك ترم مياني به صورت زیر تعریف کرد :

$E \rightarrow E + E \mid E - E \mid E / E \mid E * E \mid (E) \mid id \mid No$

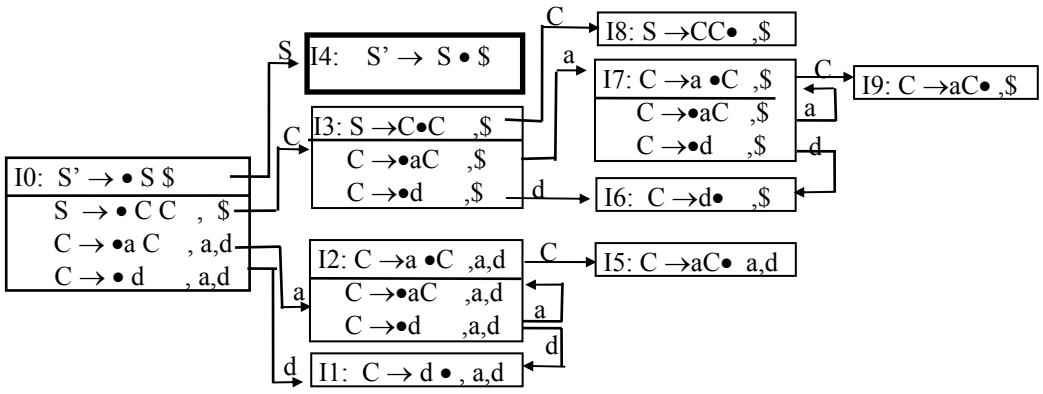
همانطور که مشاهده مي کنید در این گرامر بجاي سه ترم مياني E وFو T فقط یك ترم مياني استفاده شده و بجاي 9 قاعده براي فرم کلي عبارات ، 7 قاعده استفاده شده است که جدول را بسیار کوچکتر مینماید البته ابهام در گرامر مشکلات قابل حلي بوجود خواهد آورد که در مورد آنها کاملا" بحث خواهد شد.

روش ديگر براي كوچكتر نمودن جدول تجزيه پائين به بالا ، استفاده از گرامرهاي فرم محدودتري به اسامي SLR و LALR است. اين نوع گرامر ها در ادامه بحث توصيح داده خواهند شد.

٥.٤ گرامرهای LALR(١)

در هنگام توليد جدول تجزيه LR (1) چنانچه بتوان بدون هيچگونه مشكلي حالات با وضعيت هاي مشابه ، اما ترمهاي پيش بيني متفاوت را با يكديگر ادغام نمود ، گرامر را LALR (1) گويند . مشكل به اين صورت مي تواند باشد كه با در دست داشتن يک ترم پيش بيني يا بعبارت ديگر يك ورودي بعدي نتوان تصميم گرفت كه چه عملي بايد انجام شود. براي نمونه نتوان تصميم گرفت كه آيا عمل كاهش بايد انجام شود يا عمل انتقال . براي درك بهتر مسئله به گرامر و گراف تجزيه مربوط به آن در شكل 5.7 توجه كنيد.

0 S' → S $
1 S → C C
2 C → a C
3 C → d



شكل 5.7 - گراف تجزيه **LR (1)**

در گراف شكل 5.7 دو حالت I2 و I7 داراي وضعيتهاي يكسان با ترمهاي پيش بيني متفاوت هستند. حالات I1 و I6 و همچنين دو حالت I5 و I9 نيز به همين صورت مي باشند. اگر اين حالات را با يكديگر ادغام نماييد ، حاصل بصورت زير خواهد بود :

الف: حالت I2 با I7 ادغام شده ، حالت جديد I2_7 ايجادميشود.

I7: C →a •C ,$
C →•aC ,$
C →•d ,$
+
I2: C →a •C ,a,d
C →•aC ,a,d
C →•d ,a,d
→
I2_7: C →a •C ,$,a,d
C →•aC ,$,a,d
C →•d ,$,a,d

ب: حالت I_1 با I_6 ادغام شده ، حالت جدید I_{1_6} ایجادمیشود.

$I1: C \to d \bullet ,a,d \quad + \quad I6: C \to d \bullet ,\$ \quad \to \quad I1\_6: C \to d \bullet ,\$,a,d$

ج: حالت I_5 با I_9 ادغام شده ، حالت جدید I_{5_9} ایجادمیشود.

$I5: C \to aC \bullet ,a,d \quad + \quad I9: C \to aC \bullet ,\$ \quad \to \quad I5\_9: C \to aC \bullet ,\$,a,d$

اکنون در جدول تجزیه باید بجاي حالتهاي ادغام شده ، حالت جدید ادغامي را قرار داد. براي نمونه در بالا حالتهاي I_5 و I_9 از جدول تجزیه LR(1) حذف و با I_{5_9} جایگزین مي شوند. به این ترتیب ، هر دستورالعمل انتقال S_5 با S_9 با دستورالعمل S_{5_9} جایگزین میشود. هر عمل GoTo به این حالتها نیز به GoTo به I_{5_9} تبدیل مي شود. جدول تجزیه قبل از جایگزیني و پس از جایگزیني حالتهاي ترکیب شونده در زیر مشخص شده است:

	Action			GoTo	
	a	d	$	S	C
0	S2_7	S1_6	-	4	3
1_6	R3	R3	R3	-	-
2_7	S2_7	S1_6	-	-	5_9
3	S2_7	S1_6	-	-	8
4		Accept			
5_9	R2	R2	R2		
8			R1		

الف- پس از ادغام

	Action			GoTo	
	a	d	$	S	C
0	S2	S1		4	3
1	R3	R3			
2	S2	S1			5
3	S7	S6			8
4	Accept				
5	R2	R2			
6			R3		
7	S7	S6			9
8			R1		
9			R2		

ب- قبل از ادغام

شکل 5.8- جدول تجزیه LR(1) و LALR(1)

قبل از ادغام ، باید حالتهاي کاندید را در داخل جدول مشخص کرد. هر جا که دستورالعمل GoTo به یکي از حالتهاي ادغامي وجود دارد ، نام حالت ادغامي به جاي آن قرار مي گیرد. همین عمل را براي دستورالعمل هاي انتقال یا در اصطلاح Shift نیز باید انجام داد.

مثال - جدول تجزیه LALR(1) براي گرامر زیر ایجاد کنید:

0 $E' \to E \$$
1 $E \to E+ T$
2 $E \to T$
3 $T \to T * F$
4 $T \to F$
5 $F \to (E)$
6 $F \to id$

حالات گراف تجزیه LR(1) براي اين گرامر در زير مشخص شده است:

I0: E' → • E $ E → • E+ T , $,+ E → • T , $,+ T → •T * F , $, +, * T → • F , $, +, * F → • (E) , $, +, * F → • Id , $, +, *	I1: F → Id • , $, +, *	I2: F → (• E) , $, +, * E → • E+ T ,),+ E → • T ,),+ T → •T * F ,), +, * T → • F ,), +, * F → • (E) , }, +, * F → • Id , }, +, *
I3: T → F• , $, +,*	I4: E → T • ,$,+ T → T• * F ,$,+,*	I5: E' → E• $ E → E•+ T , $,+
I6: F → Id • ,),+,*	I7: F → (• E) ,), +, * E → • E+ T ,),+ E → • T ,),+ T → •T * F ,), +, * T → • F ,), +, * F → • (E) , }, +, * F → • Id , }, +, *	I8: T → F• ,), +,* I9: E → T • ,),+ T → T• * F ,),+,* I10: F → (E •) ,$,+,* E → E • + T ,),+
I11: T → T* • F ,$,+, * F → • (E) ,$, +, * F → • Id ,$, +, *	I12: E → E+ • T ,$, + T → •T * F , $, +, * T → • F , $, +, * F → • (E) , $, +, * F → • Id , $, +, *	I13: F → (E •) ,),+,* E → E • + T ,),+
I14: T → T* • F ,) , +, * F → • (E) , }, +, * F → • Id , }, +, *	I15: E → E+ • T ,),+ T → •T * F ,), +, * T → • F ,), +, * F → • (E) , }, +, * F → • Id , }, +, *	I16: F → (E) • ,$,+,* I17: T → T*F • ,$,+, * I18: E → E+ T• ,$, + T → T• * F , $, +, *
I19: F → (E) • ,),+,*	I20: T → T*F • ,),+, *	I21: E → E+ T• ,), + T → T• * F ,), +, *

در زير جدول تجزيه LR(1) و LALR(1) براي اين گرامر ارائه شده است.

	Action						Goto		
	id	(	)	+	*	$		T	E
0	S1	S2					3	4	5
1			R6	R6	R6	R6			
2	S1	S2					3	4	10
3			R4	R4	R4	R4			
4			R2	R2	S11	R2			
5				S12		acc			
10			S16	S12					
11	S1	S2					17		
12	S1	S2					3	18	
16			R5	R5	R5	R5			
17			R3	R3	R3	R3			
18			R1	R1	S11	R1			

ب- جدول تجزيه LALR(1) حاصل از ترکيب حالات

	Action					Goto			
	id	(	)	+	*	$	F	T	E
0	S1	S2					3	4	5
1				R6	R6	R6			
2	S6	S7					8	9	10
3				R4	R4	R4			
4				R2	S11	R2			
5				S12		acc			
6			R6	R6	R6				
7	S6	S7					8	9	13
8			R4	R4	R4				
9			R2	R2	S14				

10			S16	S15					
11	S1	S2					17		
12	S1	S2					3	18	
13			S19	S15					
14	S6	S7					20		
15	S6	S7					8	21	
16				R5	R5	R5			
17				R3	R3	R3			
18				R1	S11	R1			
19			R5	R5	R5				
20			R3	R3	R3				
21			R1	R1	S14				

شكل 5.9- جداول تجزيه LR(1) و LALR(1)

براي ادغام حالات ، ابتدا حالاتي را كه بدون ايجاد مسإله اختلال در كاهش و انتقال و به سادگي قابل ادغام هستند بايد در نظر گرفت. حالات I1 و I6 ، I3 و I8 ، I16 و I19 و بلاخره I17 و I20 بلافاصله قابل ادغام هستند. باجايگزيني حالات معادل مشاهده ميشود كه حالات I11 و I14 نيز قابل ادغام هستند. با ادغام اين دو حالت مي توان نتيجه گرفت كه دوحالت I18 و I21 نيزيكسان هستند.

۵.۵ گرامرهای SLR(۱)

گرامرهاي Simple LR(1) يا بطور مختصر SLR(1) روشي ديگر براي تجزيه پائين به بالا را ارائه مي كنند. اين گرامرها بسيار محدودتر از گرامرهاي LALR(1) و بالنتيجه گرامرهاي LR(1) هستند زيرا، در اين نوع گرامرها فرض بر اين است كه ترمهاي پيش بيني براي يك ترم مياني A وابسته به محل قرار گرفتن A در سمت راست قواعد نمي باشد. بلكه براي هر ترم مياني A ، مستقل از چگونگی قرار گرفتن آن در سمت راست قواعد مختلف، همواره مجموعه Follow(A) شاخص ترمهاي پيش بيني براي A است. بنابراين تعداد حالات و يا رديف هاي جدول تجزيه مساوي با جدول LALR(1) است زيرا ، در اينجا همانند LALR(1) ترمهاي پيش بيني عاملي براي تفكيك دو حالت از يكديگر نيستند. به عبارت ديگر نمي توان دو حالت مشخص نمود كه داراي وضعيت هاي يكسان اما ترم هاي پيش بيني متفاوت باشند.

جهت درك بهتر نقش ترمهاي پيش بيني در گرامرها به مثال زير توجه كنيد :

Statement → IfSt | WhileSt

IfSt → *If* condition *Then* Statement ElsePart

WhileSt → *While* condition *Do* Statement

با دقت در این مثال مي بينيم كه پس از مشاهده if درون IfSt. ترم مياني Condition. قرارگرفته است. درون While St. نيز پس از كلمه While ترم مياني condition ظاهر شده است امّا، در قالب جمله if بعد از Condition انتظار ديدن Then و در قالب جمله While پس از Condition انتظار مشاهده DO مي رود .بنابراين مشاهده مي شود كه وابسته به اينكه يك ترم مياني سمت راست كدام قاعده قرار گرفته ترمهاي پيش بيني آن متفاوت است. اگر ترم مياني Condition در وضعيت زير باشد :

IfSt → *If* • condition *Then* Statement ElsePart

آنگاه ترم پيش بيني براي گسترشهاي متفاوت condition كلمه Then خواهد بود. يعني بايد گسترشهاي متفاوت condition با ترم پيش بيني Then را پس از وضعيت فوق به حالت مربوطه افزود. در صورتيكه ، اگر Condition در قالب جمله While و در وضعيت زير قرار گيرد :

WhileSt → *While* • condition *Do* Statement

آنگاه ترم پيش بيني براي گسترشهای متفاوت condition كلمه Do خواهد بود. يعني بايد گسترشهای متفاوت condition با ترم پيش بيني Do را پس از وضعيت فوق به حالت مربوطه افزود.

در گرامرهاي SLR(1) گرامر آنقدر محدود است كه مهم نيست ترم مياني در داخل چه قاعده اي قرار داشته باشد . در اين نوع از گرامرها همواره براي يك ترم مياني ترمهاي پيش بيني مساوي با مجموعه Follow آن ترم مياني است. در مثال فوق اگر بجاي Then در تعريف IfSt كلمه Do بصورت زير قرار داده ميشد

Statement → IfSt | WhileSt
IfSt → *If* condition *Do* Statement ElsePart
WhileSt → *While* condition *Do* Statement

آنگاه گرامر SLR(1) مي بود. براي نمونه در زير جدول تجزيه SLR (1) براي گرامر عبارات ايجاد شده است. در شكل 5.9 جدول هاي تجزيه LR(1) و LALR(1) براي عبارات مشخص شد. در اينجا گرامر عبارات بصورت زير مطرح است :

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

درمورد گرامرهاي SLR(1) قبل از ايجاد گراف تجزيه بايد اقدام به توليد مجموعه پيرو[3] يا در اصطلاح Follow براي ترمهاي مياني نمود تا در واقع ترمهاي پيش بيني براي هر ترم مياني مشخص شود .
مجموعه پيرو براي ترمها درگرامر فوق بصورت زير است.

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

اكنون با در دست داشتن مجموعه پيرو براي ترمهاي مياني بسادگي مي توان گراف تجزيه SLR (1) را توليد نمود.

I0: E' → • E $ E → • E+ T E → • T T → •T * F T → • F F → • (E) F → • Id	I1: F → Id •	I2: F → (• E) E → • E+ T E → • T T → •T * F T → • F F → • (E) F → • Id	I3: T → F• I4: E → T • T → T• * F I5: E' → E• $ E → E•+ T
I6: F → (E •) E → E • + T I7: T → T*•F F → • (E) F → • Id	I8: E → E+ • T T → •T * F T → • F F → • (E) F → • Id	I9: F → (E) • I10: T → T*F • I11: E → E+ T• T → T• * F	

در زير جدول تجزيه SLR(1) براي گرامر فوق ارائه شده است . بايد توجه داشته باشيد كه براي اين دسته از گرامرها مجموعه هاي پيرو ترم هاي پيش بيني را تشكيل ميدهند. لذا، در داخل جدول تجزيه براي كاهش id به F بر طبق قاعده شماره 6 در زير ستونهاي مربوط به عناصر مجموعه پيرو F دستورالعمل R6 قرار داده شده است.

	Action						Goto		
	id	(	)	+	*	$	F	T	E
0	S1	S2					3	4	5
1			R6	R6	R6	R6			
2	S1	S2					3	4	6
3			R4	R4	R4	R4			
4			R2	R2	S7	R2			
5				S8		acc			
6			S9	S8					
7	S1	S2					10		
8	S1	S2					3	11	
9			R5	R5	R5	R5			
10			R3	R3	R3	R3			
11			R1	R1	S7	R1			

[3] همانگونه كه در فصل قبل نيز توضيح داده شد براي يك ترم مياني مجموعه پيرو براي يك ترم مياني به مجموعه ترم هايي اطلاق مي شود كه مي تواند بعد از آن ترم مياني ظاهر شود . بنابراين ، براي بدست آوردن مجموعه پيرو براي يك ترم مياني بايد به قوانين نگريست ومجموعه First براي ترمهاي بعدي آن را بدست آورد. در حالت كلي :

الف- از قاعده $A \rightarrow \alpha B \beta$ مي توان نتيجه گرفت $Follow(B) \supset \{b\}$

ب - از قاعده $A \rightarrow \alpha B \beta$ مي توان نتيجه گرفت $Follow(B) \supset First(\beta)$

ج - از قاعده $A \rightarrow \alpha B$ مي توان نتيجه گرفت $Follow(B) \supset First(A)$

شكل 5.9- جداول تجزيه SLR(1) براي گرامر ساده عبارات

٥.٦ گرامرهای مبهم

همانگونه كه قبلاً توضيح داده شد ، گرامر مبهم گرامري است كه بر اساس آن بيش از يك درخت تجزيه براي جمله داده شده ودر نتيجه دو مفهوم متناقص بتوان داشت. با استفاده از گرامرهاي مبهم مي توان اندازه جدول تجزيه پائين به بالا را كوچكتر نمود. براي نمونه گرامر ساده عبارات را كه در بخشهاي قبل جدول تجزيه LR(1) ، LALR(1) و SLR(1) براي آن ايجاد شد را بخاطر بياوريد. جدول تجزيه SLR(1) و LALR(1) براي اين گرامر داراي يازده حالت و در مجموع 108خانه از حافظه را اشغال ميكرد. با نوشتن گرامر عبارات به صورت مبهم زير:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid NO \mid id$$

تعداد ترم هاي مياني كاهش مي يابد و نشان داده خواهد شد كه . امّا، مبهم بودن گرامر و نوشتن آن به صورت فوق دو مشكل ايجاد مي كند . اين مشكلات توضيح داده خواهند شد.

الف- در گرامر مبهم فوق الويت عملگرها يكسان است.

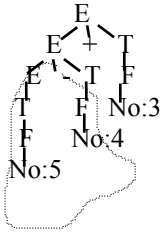
ب- اجتماع عملگرهاي جمع و ضرب مشخص نيست.

قواعد مربوط به گرامر عبارات در حالت غير مبهم بيانگر الويت و اجتماع عملگرها است.

0 $E' \rightarrow E \$$
1 $E \rightarrow E+ T$
2 $E \rightarrow E- T$
3 $E \rightarrow T$
4 $T \rightarrow T * F$
5 $T \rightarrow T / F$
6 $T \rightarrow F$
7 $F \rightarrow (E)$
8 $F \rightarrow id$
9 $F \rightarrow No$

در گرامر فوق + و - داراي اجتماع چپ هستند. براي نمونه درخت تجزيه براي عبارت زير را در نظر بگيريد :

5 - 4 + 3



همانگونه که مشاهده میکنید ابتدا باید عبارت 4 - 5 را محاسبه نمود تابتوان پس از آن حاصل را با 3 جمع کرد. در واقع از چپ به راست عمل جمع و تفریق انجام میشود. این بخاطر بیان قواعد مربوط به E بصورت خودبازگشتی چپ است. اگر قواعد مربوط به E بصورت خودبازگشتی راست بیان میشد. یعني بصورت :

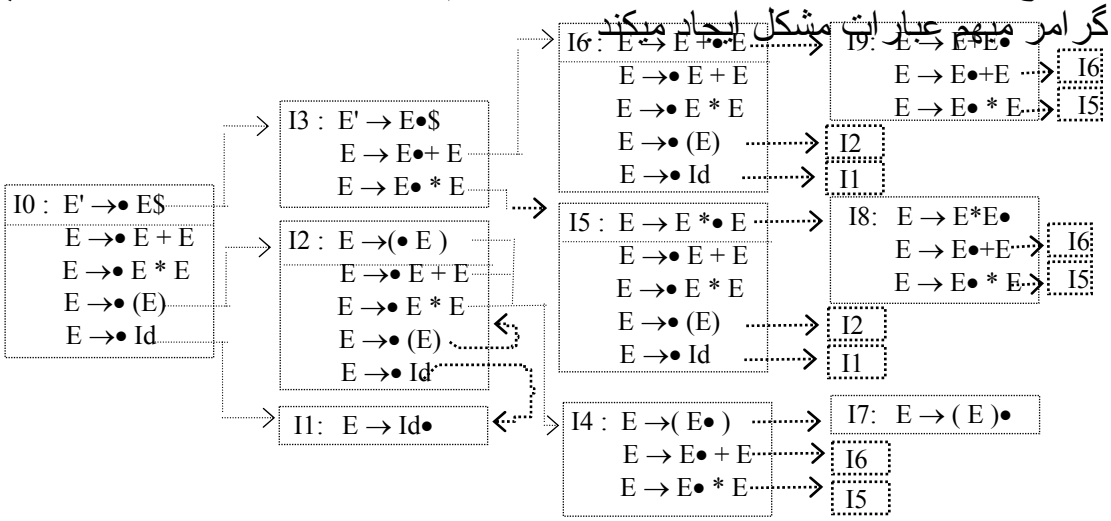
$$E \rightarrow T+ E \mid T - E \mid T$$

آنگاه عبارت 3 + 4 - 5 از راست به چپ محاسبه مي شد. یعني ، ابتدا 3 + 4 و سپس 5 - 7 محاسبه میشد. بر طبق گرامر فوق قبل از اینکه بتوان E را انجام داد باید T را محاسبه کرد. در ون T عملیات ضرب و تقسیم و درون E عملیات تفریق و جمع نهفته است. لذا، ضرب و تقسیم بر طبق این گرامر بر جمع و تفریق الویت دارند.

بنابراین با بیان گرامر خلاصه عبارات بصورت مبهم :

0 $E' \rightarrow E\$$
1 $E \rightarrow E + E$
2 $E \rightarrow E * E$
3 $E \rightarrow (E)$
4 $E \rightarrow Id$

اولاً الوییها یکسان و ثانیاً اجتماع عملگرها نامشخص خواهد بود. نامشخص بودن اجتماع و یکسان بودن الویت عملگرها در هنگام ایجاد جدول تجزیه SLR(1) براي گرامر مبهم عبارات مشکل ایجاد میکند



جدول تجزیه براي گراف فوق در شکل 9.10 داراي دو نوع عملکرد براي عملگرهاي + و * در حالات 8 و 9 است. در این دو حالت به اختلال در کاهش و انتقال وجود دارد و مشخص نیست که با مشاهده دو عملگر + و * آیا عمل کاهش را باید انجام داد یا انتقال. باید توجه داشته باشید که در گرامرهاي SLR(1) ترمهاي پیش بیني را مجموعه هاي پیرو مشخص میکنند. در گرامر فوق براي ترم میاني E باید مجموعه پیرو محاسبه شود.

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

	Action						Goto
	id	(	)	+	*	$	E
0	S1	S2					3
1			R4	R4	R4	R4	
2	S1	S2					5
3				S6	S5	Accept	
4			S7	S6	S5		
5	S1	S2					8
6	S1	S2					9
7			R3	R3	R3	R3	
8			R2	~~S6~~, R2	~~S5~~, R2		
9			R1	~~S6~~, R1	S5, ~~R1~~		

شكل 5.10- جداول تجزيه SLR(1) براي گرامر مبهم عبارات

در حالت 8 با توجه به وضعيت "$E \rightarrow E*E\bullet$" ميتوان گفت كه قبلاً يك علامت * در ورودي ديده شده است. طبق وضعيت "$E \rightarrow E\bullet+E$" انتظار مشاهده + در سر ورودي ميرود. لذا، با مشاهده يك عملگر + در ورودي چون ضرب بر جمع الويت دارد عمل كاهش R2 انجام ميشود. براي نمونه عبارت $2 * 3 \bullet + 4$ را در نظر بگيريد. مسلماً در اينجا قبل از انتقال علامت + به داخل پشته تجزيه ، بايد عمل ضرب انجام شود لذا، بجاي S6 عمل R2 انجام ميشود .

در حالت 9 با توجه به وضعيت "$E \rightarrow E+E\bullet$" ميتوان گفت كه قبلاً يك علامت + در ورودي ديده شده است. طبق وضعيت "$E \rightarrow E\bullet*E$" انتظار مشاهده * در سر ورودي ميرود. لذا، با مشاهده يك عملگر * در ورودي چون ضرب بر جمع الويت دارد عمل انتقال S5 انجام ميشود. براي نمونه عبارت $2 + 3 \bullet * 4$ را در نظر بگيريد. مسلماً در اينجا قبل از انجام عمل جمع بايد عمل ضرب انجام شود لذا، علامت * به داخل پشته تجزيه انتقال داده شده ، بجاي R1 عمل S5 انجام ميشود.

در حالت 8 با توجه به وضعيت "$E \rightarrow E*E\bullet$" ميتوان گفت كه قبلاً يك علامت * در ورودي ديده شده است. طبق وضعيت "$E \rightarrow E\bullet*E$" انتظار مشاهده * ديگري در سر ورودي ميرود. لذا، با مشاهده يك عملگر * در سر ورودي چون ضرب داراي اجتماع چپ است ، عمل كاهش R2 انجام ميشود. براي نمونه عبارت $2 * 3 \bullet * 4$ را در نظر بگيريد. مسلماً در اينجا قبل از انتقال علامت * به داخل پشته تجزيه ، بايد عمل ضرب انجام شود لذا، بجاي S5 عمل R2 انجام ميشود .

در حالت 9 با توجه به وضعيت "$E \rightarrow E+E\bullet$" ميتوان گفت كه قبلاً يك علامت + در ورودي ديده شده است. طبق وضعيت "$E \rightarrow E\bullet+E$" انتظار مشاهده + ديگري در سر ورودي ميرود. لذا، با مشاهده يك عملگر + در سر ورودي چون جمع داراي اجتماع چپ است ، عمل كاهش R1 انجام ميشود. براي نمونه عبارت $2 + 3 \bullet + 4$ را در نظر بگيريد. مسلماً در اينجا قبل از انتقال علامت + به داخل پشته تجزيه ، بايد عمل جمع انجام شود لذا، بجاي S6 عمل R1 انجام ميشود .

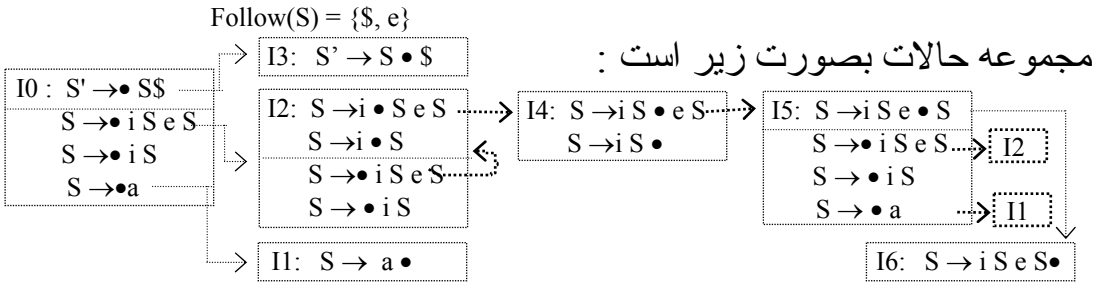
مثال - براي گرامر مبهم جملات شرطي IF جدول تجزيه SLR(1) ايجاد كنيد.

0 $S' \rightarrow S\ \$$

1 $S \rightarrow i\ S\ e\ S$
2 $S \rightarrow i\ S$
3 $S \rightarrow a$

در گرامر فوق S نمایانگر جمله ، i مخفف if و e مخفف expression است. در اولین مرحله باید مجموعه پیرو را براي ترم هاي مياني و سرترم گرامر بدست آورد. اين مجموعه در واقع ترم هاي پيش بيني را براي ترم مياني مربوطه مشخص مي كند.

مجموعه حالات بصورت زیر است :



در ردیف 4 از جدول تجزیه چون else یا e متعلق به نزدیکترین جمله if است ، S5 انتخاب شده است.

	Action				Goto
	i	e	a	$	S
0	S2		S1		3
1		R3		R3	
2	S2		S1		4
3				Accept	
4		S5, ~~R2~~		R2	
5	S2		S1		6
6		R1		R1	

شکل 5.11- جداول تجزیه SLR(1) براي گرامر مبهم جملات شرطي If

5.7 تمرین

تمرین 1 - نشان دهید که گرامر زیر LALR (1) است :

$$S \rightarrow A\,a \mid b\,a\,c \mid d\,c \mid b\,d\,a$$
$$A \rightarrow d$$

تمرین 2- آیا گرامر زیر LR(1) است.

$$S \rightarrow SAB \mid a\,B$$
$$A \rightarrow a\,A \mid A\,d$$
$$B \rightarrow B\,a \mid b\,A\,d$$

تمرین 3- آیا گرامر زیر LALR(1) است.

$$S \rightarrow a\,A\,B \mid S\,D\,b$$
$$A \rightarrow a\,D\,B \mid A\,b$$
$$B \rightarrow B\,D\,a \mid a\,b\,D$$
$$D \rightarrow D\,a \mid d\,b$$

تمرین 4- چرا جدول تجزیه LALR(1) و SLR(1) براي يك گرامر به يك اندازه هستند. براي نمونه جدول تجزیه LALR(1) و SLR(1) براي گرامر عبارات به يك اندازه است.

تمرین 5- اگر گرامري LR(1) باشد و حالات مشابه آن با یکدیگر ادغام شوند تا جدول تجزیه LALR(1) حاصل شود در هنگام ادغام حالات تنها امکان مشکل اختلال در کاهش و انتقال و نه اختلال در کاهش و کاهش و یا در اصطلاح Reduce- Reduce Conflict بوجود خواهد آمد . منظور از اختلال در کاهش و کاهش این است که براي نمونه با مشاهده a در ورودي بتوان دو عمل متفاوت R2 و R3 را انجام داد.

تمرین 6- نشان دهید که هر گرامر LL(1) يك گرامر LR(1) است ، اما بالعکس صادق نیست.

تمرین 7- با در نظر گرفتن اینکه عملگرتوان الویت بیشتري نسبت به ضرب و تقسیم دارد و اجتماع آن راست میباشد ، گرامر عبارات را تکمیل نموده ، جدول تجزیه SLR(1) براي این گرامر ایجاد کنید.

تمرین 8- در گرامر زیر علامت ↑ نمایانگر توان است. جدول تجزیه SLR(1) ایجاد کنید.

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid NO \mid id$$

تمرین 9- جدول تجزیه SLR(1) براي گرامر زیر تشکیل دهید.

$$E \rightarrow E\ \theta_1\ E \mid E\ \theta_2\ E \mid \ldots \mid E\ \theta_n\ E \mid (E) \mid id$$

فرض کنید که عملگرهاي θ_i داراي اجتماع راست و θ_i الویت بیشتري نسبت به θ_j دارد اگر اندیس i بزرگتر از j باشد.

تمرین 10- نشان دهید که گرامر زیر $LL(1)$ است و $SLR(1)$ نیست.

$$S \rightarrow A\ a\ A\ b \mid B\ b\ B\ a$$
$$A \rightarrow \varepsilon$$
$$B \rightarrow \varepsilon$$

آیا گرامر فوق $LR(1)$ است.

راهنمايي - توجه داشته باشید در هر حالتي که گسترش تهي براي ترمهاي مياني A یا B وجود داشته باشد ، مي توان در داخل جدول تجزیه در زیر ستون مربوط ترمهاي پیش بیني براي این دو ترم عمل کاهش قرار داد.

تولید کد میانی

کد میانی چیست ؟

کدی است در سطح زبان ماشین، اما مستقل از ساختار هر ماشین خاص، در واقع یك شبه اسمبلی ست، که متعلق به هیچ ماشین خاصی نیست، لذا به آن ماشین مجازی (Virtual Machine) نیز گفته می شود، برای مثال JVM که همان ByteCode است که :

- امکان پیاده سازی راست و بدون اشکال دستورالعمل های جاوا را به شبه اسمبلی فراهم نمود.

(در پاسکال Pcode و در C، Ccode داریم)

علت استفاده از کد میانی چیست؟

کد میانی :

- قابل اجرا در روی هر کامپیوتری می باشد (Portableity).
- بهینه سازی خوبی دارد (Optimization).

انواع کدهای میانی :

۱. **دستورالعملهای شبه اسمبلی.**

در واقع این دستورالعملها بیانگر یك ماشین مجازی می باشند، زیرا محدودیت در تعداد ثباتها را ندارند.

۲. **دستورالعملهای سه آدرسه.**

۳. **درختهای خلاصه نحوی.** Abstrac syntax Trees

روش تولید کد میانی :

۱. آیا در مرحله ای جداگانه کد میانی ایجاد می شود (از لحاظ سرعت بسیار کند می باشد).

۲. آیا همگام با تحلیل نحوی، این کار صورت می گیرد (از لحاظ پیاده سازی دارای مشکل می باشد).

حل مشکل :

پاسخ در روشی ست تحت عنوان ترجمه هدایت شده همگام با تحلیل نحوی(Syntax Directed Translation).

روش :

- بعد از تشخیص هر ترم میانی.
- بعد از Reduce در روش بالا به پایین.
- بعد از فراخوانی روال هر ترم میانی در روش بالا به پایین، کد مربوطه تکمیل می شود.

مثال :

تولید کد میانی برای عبارت زیر :

```
a*(b+c)+d*e
```

```
mov T1, a
mov T2, b
add T2, c
mul T1, T2
mov T2, d
mul T2, e
add T2, T1
```

هدف :

تولید مولد کد شبه اسمبلی برای عبارات چهار عمل اصلی.

```
E ➔ E + T | E - T | T
T ➔ T * F | T /  F | F
F ➔id | no | '( ' E ' )'
```

برای اینکه بتوان تجزیه گر کاهینه بازگشتی برای گرامر عبارات ایجاد نماییم، ابتدا باید گرامر را به فرم LL(1) تبدیل می کنیم.

```
E ➔  T{( + | - ) T}
T ➔  F{( * |  /)  F}
F ➔  id | no | '( ' E ' )'
```

هر بار که New Temp اجرا می شود، String جدیدی را بعنوان متغیر کمکی بر می گرداند.

```
Var
  Target, Source : Text;
  TempNo : integer;      //مقدار اولیه صفر

  TempVar : string;      //شامل رشته موقتی متغیر

Procedure  NewTemp;  //متغیر موقتی جدید ایجاد می کند

Begin
  Tempno := TempNo + 1;
  TempVar := 'T' + str( TempNo );
End;
Procedure  RemTemp;  //متغیر موقتی را برگشت می دهد

Begin
  Tempno := TempNo - 1;
End;
Procedure  Emitln( string s );
```

```
Begin
  Writeln( Target, S);
End;
Procedure  Emit( string s );
Begin
  Write( Target, S);
End;
```

حال شروع به نوشتن تك تك تابع می کنیم :

```
Stops : set of symbols;
//-------------------------------------------------------------------------------
-----------------
(*F ➔  id | no | '( ' E ' )' *)
Procedure F ( Stop : Stops; Var Resf : string; R : integer;);
Var
  R  : integer;
  ResF : string;
Begin
  R := 0;
  If ( CurrentSymbol = S-id ) or ( CurrentSymbol = S_no) then
    Begin
      ResF := CurrentToken.Lexeme;
      NextSymbol;
    End
  Else
    Begin
      Expect( S_OpenPar, Stop + [S_id, S_no, S_OpenPar]);
      E ( Stop + [S_ClosePar], ResF, R);
      Expect( S_ClosePar, Stop);
    End;
End;
//-------------------------------------------------------------------------------
-----------------
(*  T ➔  F{( * |  /)  F}  *)
Procedure  E ( Stop : Stops; Var ResT : string; R : integer;);
Var
  OpCode : Symbols;
  Operand : string;
  R1 : integer;
Begin
  F( Stop + [S_mul, S_div], ResT, R);
  While ( CurrentSymbol = S_div) or ( CurrentSymbol = S_mul) do
    Begin
      OpCode := CurrentSymbol;
      NextSymbol;
      If R = 0 then
        Begin
          NewTemp;
          OpCode := TempVar;
          Emitln('mov ' + Operand + ',' + ResT);
          R := 1;
          ResT := Operand;
```

```
        End;
      F ( Stop + [ S_mul, S_div], Operand, R1);
      If (OpCode = S_mul) then
        Emit('mul ')
      Else Emit('div ');
      Emitln(ResT + ',' + Operand);
      If R1 = 1 then
        RemTemp;
    End;
End;
//-------------------------------------------------------------------------------
-----------------
(*  E ➔  T{( + | - ) T}  *)
Procedure E ( Stop : Stops; Var ResE : string; R : integer;);
Var
  OpCode : Symbols;
  Operand : string;
  R1 : integer;
Begin
  T( Stop + [S_add, S_sub], ResT, R);
  While (CurrentSymbol = S_add) or (CurrentSymbol = S_sub) do
    Begin
      OpCode := CurrentSymbol;
      If R = 0 then
        begin
          NewTemp;
          Operand := TempVar;
          Emitln('mov ' + Operand +  ',' + ResE);
          R = 1;
          ResE := Operand;
      End; { end of  if }
      If OpCode = S_add then
        Emit('add ')
      Else Emit('sub ');
      T(Stop + [S_add, S_sub], Operand, R1);
      Emitln(ResE + ',' + Operand);
      If R1 = 1 then
        RemTemp;
    End; { end of  while }
End; { end of  procedure }
```

هدف :

تولید دستورالعملهای شبه اسمبلی برای انواع جملات.

روش :

ترجمه همراه با هدایت نحوی (Syntax Directed Translation)

```
  (* id := E *)
Procedure Assignment( Stop : Stops; Var ResA : string; R : integer);
Var
```

```
  ResE : string;
Begin
  ResA := CurrentToken.Lexeme;
  Expect(S_id, Stop + [S_assign]);
  Expect(S_assign, Stop + [S_id, S_no, S_OpenPar]);
  E(Stop, ResE);
  Emitln('mov ' + ResA + ',' + ResE);
  If R = 1 then
  RemTemp;
  R := 0;
End;
```

مثال :

الف-جملات تخصیصی

ب-جملات شرطی (جملات If)

```
If a > b then c :=  a - b
Else c := a + b;

mov T1, a
cmp T1, b
jle L1
mov T1, a
sub T1, b
mov c, T1
jmp L2
L1:
mov T1, a
add T1, b
mov c, T1
L2:

Var
  LabelNo : integer;
Function NewLabel : string;
Begin
  LabelNo := LabelNo + 1;
  NewLabel := 'L' + str(LabelNo);
End;

(* Condition ›    E Relop E *)
Procedure Condition( Stop : Stops; Var ResC : string; R : integer);
Var
  ResE , ResE1 : string;
  LFalse : string;
  R1, R2 : integer;
  OpCode : Symbols;
Begin
  E(Stop + [S_lt, S_le, S_gt, S_ge, S_ne, S_e], ResE, R1);
  OpCode := CurrentSymbol;
  Relop(Stop + [ S_id, S_if,…]);
```

```
  E(Stop, ResE1, R2);
  Emitln('cmp ' + ResE + ',' + ResE1);
  If R1 = 1 then
    RemTemp;
  R := 1;
  ResC := NewTemp;
  Emitln('xor ' + ResC + ',' + ResC);
  LFalese := NewLabel;
  Case OpCode of
    S_lt :
      Emitln('jge ' + LFalse);
    S_le :
      Emitln('jg ' + LFalse);
    S_ne :
      Emitln('je ' + LFalse);
    …
  end;
  Emitln('inc ' + ResC);
  Emitln(LFalse + ':');
End;
```

تمرین :

برای عبارات If ، Case و While با در نظر گرفتن گرامر Condition ، کد شبه اسمبلی تولید نمایید.

```
(* Ifst ›    If  Condition Then  st  Else st *)
Procedure Ifst( Stop : Stops; Var ResIf : string; R : integer);
Var
  ResCond : string;
  LabelElse, LabelEnd : string;
  RC : integer;
Begin
  Expect(S_if, Stop + [S_OpenPar, S_id, S_no, S_not]);
  Condition(Stop + [S_then], ResCond, RC);
  Emitln('cmp ' + ResCond + ',' + '0');
  LabelElse := NewLabel;
  RemTemp;
  Emitln('je ' + LabelElse);
  Expect(S_then, Stop + [S_id, S_while, S_begin, …]);
  st(Stop + [S_Else], '0', 0);
  if CurrentSymbol = S_Else then
    Begin
      LabelEnd := NewLabel;
      Emitln('jmp ' + LabelEnd);
      Emitln(LabelElse + ':');
      NextSymbol;
      st(Stop, '0', 0);
    End
  Else
    LabelEnd := LabelElse;
  Emitln(LabelEnd + ':');
End;
```

مثال :

```
If a > b then  c := a + 1
Else c := b + a;
  xor T1, T1
  mov T2, a
  cmp T2, b
  jle L1
  inc T1
L1:
  cmp T1, 0
  je L2
  mov T1, a
  add T1, 1
  mov c, T1
  jmp L3
L2:
  mov T1, b
  add T1, a
  mov c, T1
L3:
  …
```

نکته :

باید توجه نمود که در تابع Condition که قبلا نوشته شده است، باید کد بصورتی تولید شود که ابتدا متغیر موقتی به ResC تخصیص داده شود تا پس از انجام مقایسه بتوان به درستی RemTemp را به اجرا در آورد.

تمرین :

رویه هایی بنویسید که برای جملات Case کد تولید نمایند.

```
Case› Case id of
            CasePart {CasePart}
            [ElsePart]
          End;
Casepart› (id  | no ) {, ( id | no )} ':' st ';'
```

تولید کد اسمبلی

هدف :

استفاده از دستورالعملهای اسمبلی پروسسورهای اینتل جهت تولید کد اسمبلی.

در این راستا می بایست قوانین اسمبلی رعایت شود و در عملیاتی مانند جمع و ضرب از اکومولاتور (ثبات AX) به عنوان اولین عملگر استفاده شود.

محدودیت دیگر، تعداد ثباتها می باشد که محدود هستند، دیگر اینکه همواره اولین عملگر می بایست در یك ثبات قرار گیرد. برای چگونگی تخصیص ثباتها از تابعی به نامRegister Management استفاده می شود.

Register Management :

RM یا مدیریت ثباتها، یك جدول به نام Register Table می باشد که دارای وضعیت تخصیصی Registerها را در آن مشخص می کنیم. این کلاس در تابع اصلی به نام Get و Free دارد که جهت گرفتن و با آزاد کردن Registerها مورد استفاده Code Generator قرار می دهد.

یك نکته ای که RM در نظر می گیرد علاوه بر تخصیص ثبات مشخص می کند که ثبات حافظه مقدار کدام متغیر است.

```
mov DI, 1
add AX, DI
add AX, 1
mov DI, AX
mov AX, DI
mul AX, 2
mov SI, AX
Reg := GetReg('AX');
Reg2 := CheckReg(CurrentToken.Lexeme);
Reg := GetReg('j');
Reg := RM.GetReg(CurrentToken.Lexeme);
'mov DI, 1'
Reg := RM.GetReg('AX');
Reg2 := CheckReg(CurrentToken.Lexeme);
'mov AX, DI'
'add AX, 1'
Reg := GetReg(CurrentToken.Lexeme);
'mov DI, AX'
```

تا به حال برای تولید کدهای اسمبلی از متغیرهای موقتی استفاده می کردیم ، برای مثال

```
(a+b*c) * (d+c*f)

Mov  t1,a
Mov  t2,b
Mul  t2,c
Add  t1,t2
Mov  t2,d
Mov  t3,c
Mul  t3,f
Add  t2,t3
Mul  t1,t2
```

اما چنانچه که میدانیم کدهای بالا به صورت عملی کاربردی ندارد و اجرا نخواهد شد. برای اینکه بتوانیم کدهـای بـالا را قابل اجرا کنیم باید به جای متغیرهای موقتی از ثباتها استفاده کنیم به این طدیق که به جای متغیرهای موقـتی ، ثباتهـا را جایگزین کنیم.

سئوال: چرا نیاز به مدیریت ثباتها داریم؟

۱) تعداد ثباتها به اندازه تعداد متغیرهای موقتی نیست(محدودیت ثباتها)

۲) نسبت دادن متغیر های موقتی به ثباتها مشکل بوجود می آورد.

یک روش کلی (جایگزاری):

برای تبدیل کد بالا به کد قابل اجرا باید جای متغیرهای موقتی از ثباتها استفاده کنیـم. بـرای ایـن کـار ابتـدا کـد بـرای متغیرهای موقتی ایجاد کرده و سپس در تبدیل به جای هر متغیر موقتی ثبات قرار می دهیم . در این اینجا دو سئوال پیش می آید :

۱) تعداد ثباتها محدود است .

۲) چه وقت نیاز به آن ثباتی که استفاده شده است نداریم؟

جواب ۱) فرض کنید ما دو ثبات داریم، داخل یکی مقدار T1 را قرار دادهایم وداخل دیگری T2را.حال بـه یـك ثبـات دیگر نیاز داریم، می توانیم بسته به قراردادی که برای خودمان در نظر می گیریم یکی از ثباتها را برای انجام کارمان خالی کنیم یعنی مقدارش را در یك متغیر محلی به نام X1قرار دهیم . سپس از آن ثبات استفاده کنیم در اینجـا مـا از FIFO استفاده می کنیم ، یعنی اینکه آن ثباتی که دارای شماره کمتری از نظر الویت است مقدارش را در متغیر موقـتی X1 قـرار می دهیم که خواهیم داشت :

ثبات	الویت
Ax -> T1	**1**
Bx -> T2	**2**

الویت Ax کمتر است :

X1 -> Ax
Ax -> T3 **3**

برای انجام این کار به صورت عملی به رکوردهای زیر نیاز داریم :

رکورد RegTableRow :

۱) نام ثبات (RegName)--› از نوع String

۲) نام متغیر (VarName) --› از نوع String

۳) شماره الویت (Turm) --› از نوع Integer

۴) آیا ثبات در حال استفاده هست ؟ (Turn) --› از نوع Boolean

برای هر متغیر که به آن یك ثبات اختصاص میدهیم رکورد بالا را پر میکنیم ، برای مثال :

Ax -> T1

خواهیم داشت :

RegName = Ax
VarName = T1
Turn = 1
Busy = True

برای زمانی که ثبات کم می آوریم ، باید یك ثبات را خالی کنیم . یك رکورد دیگر باید داشته باشیم .

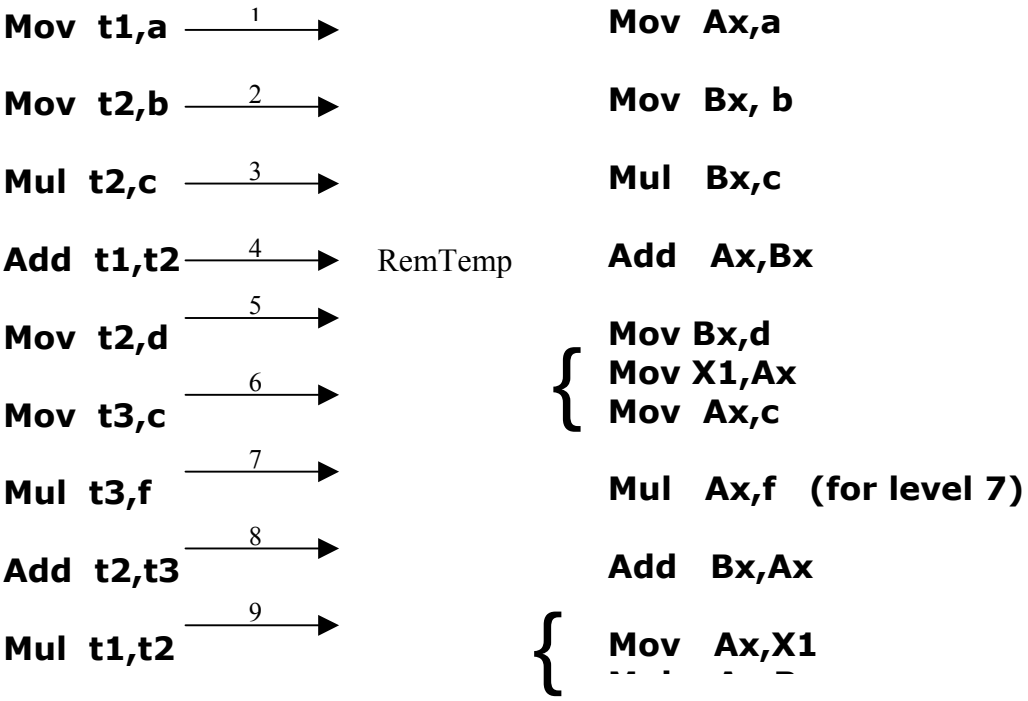
رکورد TempTableRow :

۱) نام متغیر موقتی (TempName) --› از نوع String

۲) نام متغیر یا ثبات (varName) --› از نوع String

به مثال زیر توجه کنید :

(a+b*c) * (d+c*f)



در مورد کد بالا به موارد زیر توجه کنید :

۱) ما فقط دو ثبات داریم : AX,BX

۲) به انجام مراحل زیر توجه کنید :

مرحله ۱ :

Regname=AX
Varname=T1
Turn=1
Busy=True

داخل جدول ثباتها و متغیرها را جستجو میکنیم ,
اگر به T1 ثبات اختصاص نـداده باشـیم , بـه آن
ثبات اختصاص می دهیم.

مرحله ۲ :

داخل جدول ثباتها و متغیرها را جستجو میکنیم ,
اگر به T1 ثبات اختصاص نـداده باشـیم , بـه آن
ثبات اختصاص می دهیم.

Regname=BX
Varname=T2
Turn=2
Busy=True

مرحله ۳ :

در داخل جدول ثباتها جستجو می کنیم چون به T2 یکبار ثبات داده شده است از همان در این مرحله استفاده میکنیـم .

مرحله ۴ :

در داخل جدول ثباتها جستجو می کنیم چون به T2 , T1یکبار ثبات داده شده است از همان در ایـن مرحلـه اسـتفاده میکنیم .

نکته : وقتی یك متغییر در سمت راست آورده می شود در هر مرحله أی که باشیم آن متغیـیر را در داخـل جـدول ثباتـه جستجو می کنیم اگر آن متغییر دارای یك ثبات باشد ، اشغالی آن ثبات را False می کنیـم چـون دیـگر نیـازی بـه آن نیست ، چون مقدارش در متغییر سمت چپ ریخته شده است (کاملا مثل قبل عمل می کنیم در قبل ما Remteoup مـی کردیم) .

باید یك رجیستر پیدا کنیم که خالی باشد تا مقدار D را در آن قرار دهیم داخل جدول رجیستر جستجو می کنیـم چـون bx ، Busy اش False است می یوانیم از این ثبات برای قرار دادن d استفاده کنیم .

باید داخل جدول ثباتها جستجو کنیم تا یك ثبات خالی پیدا کنیم ولی چون دیگر ثبات خالی دیگری وجود ندارد مجبوریم یکی از ثباتها را به یك متغییر موقتی بریزیم و از آن ثبات خالی شده استفاده کنیم ، برای این کار به صورت زیر عمل می کنیم .

(1) داخل جدول متغییر ها

Temp name = x1
Vov name = T1
X1 ← ax

(2) **Reg name = ax**

Vav name = T3
Turn = 3
Busy = True

مانند قبل عمل می کنیم داخل جدول ثباتها چون به T3 یك بار ثبات اختصاص داده شده اسـت دیـگر بـرای آن ثبـات جدیدی در نظر نمی گیریم و عمل ضرب را انجام می دهیم .

عمل جم++ع انجام می شود ولی چون ax در سمت راست قرار دارد دیگر نیازی به آن نیســت و (Busy = False) اشغال آنرا برابر False قرار می دهیم (یعنی می توان از این استفاده کرد) .

در مرحله (q) باید T1, T2 را جستجو کنیم T2 را در جدول مربوط به ثباتها پیدا کرده و مقدارش که bx را در نظـر می گیریم ولی T1 در جدول ثباتها وجود ندارد پس جدول مربوط به متغییر ها را نیز جستجو می کنیم آنرا پیدا می کینم

می فهمیم مقدارش در x1 وجود دارد آنرا داخل یك ثبات خالی قرار می دهیم (اگر وجود نداشت یك ثبات را خالی می کنیم) و سپس عمل Mul را انجام میدهیم .