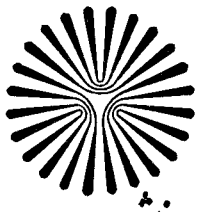
دانشگاه پیام نور

اصول طراحی کامپایلرها

مهندس جعفر پورامینی

درسنامه



دانشگاه پیام نور

اصول طراحی کامپایلرها

(رشتهٔ کامپیوتر)

مهندس جعفر پورامینی

گروه کامپیوتر
(۱۲/د)

سرشناسه	:	پورامینی، جعفر
عنوان و پدید آور	:	اصول طراحی کامپایلرها (رشتهٔ کامپیوتر) / مؤلف: جعفر پورامینی
مشخصات نشر	:	تهران: دانشگاه پیام‌نور، ۱۳۸۴.
مشخصات ظاهری	:	۳۱۴ص.
فروست	:	دانشگاه پیام نور؛ ۱۱۷۳. گروه کامپیوتر؛ ۱۲/ د
شابک	:	978 - 964 - 387 - 196 - 3
وضعیت فهرست نویسی	:	فیپا.
موضوع	:	۱. آموزش از راه دور – ایران .
موضوع	:	۲. کامپایلرها(برنامه کامپیوتر)- آموزش برنامه‌ای. ۳. کامپایلرها(برنامه کامپیوتر)- آزمونها و تمرینها.
شناسه افزوده	:	الف. دانشگاه پیام نور. ب. عنوان.
رده بندی کنگره	:	۹۶پ ۹ الف/LC۵۸۰۸
رده بندی دیویی	:	۳۷۸/۱۷۵۰۹۵۵
شمارهٔ کتابشناسی ملی	:	۸۴م-۲۳۲۲۱

دانشگاه پیام نور

اصول طراحی کامپایلرها

مؤلف: مهندس جعفر پورامینی

ویراستار علمی: مهندس سید ناصر آیت

تهیه و تولید: مدیریت تولید مواد و تجهیزات آموزشی

لیتوگرافی، چاپ و صحافی: انتشارات دانشگاه پیام‌نور

شمارگان: ۵۰۰۰ نسخه

نوبت و تاریخ چاپ: چاپ اول مرداد ۱۳۸۵، چاپ پنجم اردیبهشت ۱۳۸۹

شابک: ۳ - ۱۹۶ - ۳۸۷ - ۹۶۴ - ۹۷۸

ISBN: 978 - 964 - 387 - 196 - 3

قیمت: ۲۵۰۰۰ ریال

بسم الله الرحمن الرحیم

پیشگفتار ناشر

کتابهای دانشگاه پیام نور حسب مورد و با توجه به شرایط مختلف به صورت درسنامه، آزمایشی، قطعی، متون آزمایشگاهی، فرادرسی، و کمک‌درسی چاپ می‌شود. کتاب **درسنامه** (د) نخستین ثمرهٔ کوششهای علمی صاحب اثر است که براساس نیازهای درسی دانشجویان و سرفصلهای مصوب تهیه می‌شود و پس از داوری علمی در گروههای آموزشی، بدون طراحی آموزشی و ویرایش چاپ می‌شود. با تجدیدنظر صاحب اثر و دریافت بازخوردها و اصلاح نارساییها، درسنامه با طراحی آموزشی، ویرایش، و طراحی فنی ـ هنری به صورت **آزمایشی** (آ) چاپ می‌شود. با دریافت نظرهای اصلاحی، صاحب اثر در کتاب تجدید نظر می‌کند و کتاب به صورت **قطعی** (ق) چاپ می‌شود. در صورت ضرورت، در کتابهای چاپ قطعی نیز تجدید نظرهای اساسی به عمل می‌آید یا متناسب با پیشرفت علوم و فناوری بازنویسی می‌شوند. **متون آزمایشگاهی** (م) متونی است که دانشجویان با استفاده از آن و راهنمایی مربیان کارهای عملی آزمایشگاهی را انجام می‌دهند. کتابهای **فرادرسی** (ف) و **کمک‌درسی** (ک) به منظور غنی‌تر کردن منابع درسی دانشگاهی تهیه می‌شوند. کتابهای فرادرسی با تأیید معاونت پژوهشی و کتابهای کمک‌درسی با تأیید شورای انتشارات تهیه می‌شوند.

مدیریت تدوین

فهرست

پیشگفتار

اصول طراحی کامپایلر یکی از جذاب ترین مباحث علم کامپیوتر است که شامل مباحث تئوری و عملی است. در این کتاب سعی شده است جنبه های تئوری و عملی مورد توجه و بررسی قرار گیرد تا درک مباحث تئوری ساده تر گشته و کاربرد آن عینی تر شود.

مطالب همراه با مثالهای متعدد ارائه گردیده است و سعی شده است حل مثالها مرحله به مرحله به همراه شکلها و جداول لازم ارائه شود. امید است، دانشجو با توجه به متن و حل مرحله به مرحله مثالها، مطالب را به خوبی درک کند. در آخر هر فصل نیز تمرینات متعدد جهت خودآزمایی دانشجویان ارائه گردیده است. با توجه به اهمیت انجام پروژه های برنامه نویسی برای درک مطالب، در هر فصل تمرینات برنامه نویسی نیز ارائه گردیده است. علاوه بر مثالهای درون فصلها و تمرینات آخر هر فصل، تستهای تکمیلی همراه کلید آنها در آخر کتاب ارائه گردیده است.

بدیهی است، کتاب حاضر خالی از اشکال نیست بنابراین هر گونه اظهار نظر و پیشنهادی که باعث بهبود کیفیت کتاب شود مورد قدردانی و تشکر خواهد بود(pouramini@yahoo.com).

لازم است از زحمات کلیه دوستان ، اساتید و دانشجویان که در تهیه این کتاب یاری کردند تشکر نمایم. علی الخصوص از دکتر کریم‌زادگان و دکتر فراهی و همچنین از آقایان یوسف خانی، تنها، شریفی، نیک مهر، قنبری، رمزی، شریفیان و به خصوص آقای یارمحمدی تشکر می کنم.

جعفر پورامینی

فصل اول

مقدمه‌ای بر کامپایلر

هدف کلی

آشنایی با مفاهیم اولیه کامپایلر و ساختار کلی یک کامپایلر

هدفهای رفتاری

پس از مطالعه این فصل انتظار می‌رود دانشجو بتواند:

۱- انواع زبانهای برنامه سازی را معرفی کند.

۲- روشهای ترجمه و اجرای برنامه به همراه مزایا و معایب هر یک را بیان کند.

۳- کامپایلر و مفسر را تعریف کند.

۴- جلوبندی و عقب بندی کامپایلر را بیان کند.

۵- ساختار کامپایلر را بیان کند.

۶- تاریخچه کامپایلر را بیان کند.

۷- ویژگیهای یک کامپایلر خوب را بیان کند.

۱-۱ مقدمه

نرم افزارها به وسیله زبانهای برنامه‌نویسی ساخته می‌شوند. زبانهای برنامه نویسی انواع مختلفی دارند که برخی از آنها عبارتند از:

۱- زبان ماشین: زبانی که در آن داده‌ها و دستورالعمل‌ها به صورت کدهای باینری (صفر و یک) نمایش داده می‌شوند و تنها زبانی است که کامپیوتر درک می‌کند، هر برنامه، باید قبل از اجرا، به زبان ماشین ترجمه شود. هر نوع کامپیوتری زبان ماشین مخصوص به خود دارد. برنامه‌نویسی به زبان ماشین بسیار مشکل و زمانبر است زیرا درک

دنباله‌ای از صفر و یک‌ها برای انسان بسیار مشکل است، به همین جهت از این زبان به جز در موارد خاص استفاده نمی‌شود.

۲- **زبان اسمبلی**[1]: زبان اسمبلی به جای کدهای باینری از کلمات اختصاری استفاده می‌کند. خوانایی برنامه‌های به زبان اسمبلی بیشتر از برنامه‌های به زبان ماشین است. برنامه‌نویسی به زبان اسمبلی نیز مشکل است، ولی از زبان ماشین ساده‌تر است. برای تبدیل برنامه اسمبلی به زبان ماشین از نرم افزار مترجمی به نام اسمبلر استفاده می‌شود، که کلمات اختصاری را به زبان ماشین ترجمه می‌کند.

۳- **زبان‌های سطح بالا:** زبان‌های سطح بالا، به زبان محاوره‌ای نزدیکترند و دارای ساختارها و دستورات بیشتر و قدرتمندتر نسبت به زبان اسمبلی هستند. از جمله این زبان‌ها می‌توان به C، پاسکال و بیسیک اشاره کرد. برنامه‌های نوشته شده با این زبانها مستقیما قابل اجرا روی ماشین نیستند. برنامه‌های به زبان سطح بالا توسط کامپایلرها[2] به زبان ماشین ترجمه می‌شود تا قابل اجرا بر روی رایانه شوند، و یا به وسیله مفسرها[3]، اجرا می‌شوند.

تفاوت اسمبلر و کامپایلر زبانهای سطح بالا در این است که اسمبلر هر دستور اسمبلی را فقط به یک دستور زبان ماشین ترجمه می‌کند در حالیکه کامپایلرها، هر دستور زبان سطح بالا را ممکن است به چندین دستور زبان ماشین ترجمه کنند. به عبارت دیگر، رابطه دستورات زبان اسمبلی به دستورات زبان ماشین یک به یک است، در حالیکه رابطه دستورات زبان سطح بالا به دستورات زبان ماشین، یک به چند است. نوع خاصی از اسمبلرها به نام ماکرو اسمبلرها[4] امکان تعریف ماکرو را به کاربران می‌دهند. هر ماکرو با دستورات دیگر اسمبلی جایگزین می‌گردد. پس از جایگزینی ماکروها با دستورات اسمبلی ترجمه برنامه به زبان ماشین انجام می‌شود. بنابراین در این نوع اسمبلرها نیز رابطه دستورات اسمبلی با دستورات ماشین یک به یک است.

۱-۲ روشهای ترجمه و اجرای برنامه‌های سطح بالا

برنامه‌های نوشته شده به زبانهای سطح بالا، مثل C ،بیسیک و پاسکال، مستقیما روی ماشین اجرا نمی‌شوند. برای ترجمه و اجرای این برنامه‌ها دو روش عمده وجود دارد که عبارتند از:

۱- استفاده از مفسر

۲- استفاده از کامپایلر

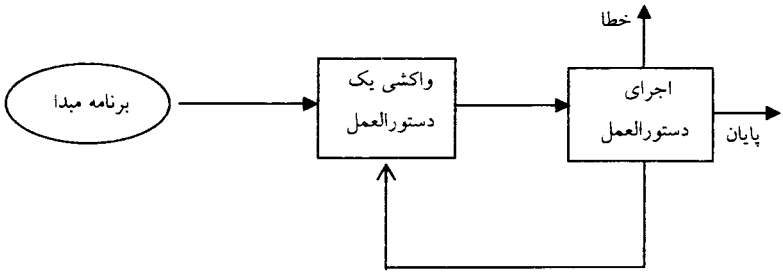
---

1. Assembly
2. Compiler
3. Interpreter
4. Macro Assembler

در بخشهای بعدی به شرح مختصر هر یک از روشهای ذکر شده می‌پردازیم.

۱-۲-۱ استفاده از مفسر

در این روش دستورالعملهای برنامه یک به یک توسط نرم افزاری به نام مفسر خوانده شده و اجرا می‌گردد. عملکرد مفسر را می‌توان به عملکرد یک شخص مترجم که در مصاحبه‌ها و مذاکرات مسئول ترجمه است تشبیه کرد. در مصاحبه‌ها و مذاکرات بین نمایندگان کشورهای مختلف مترجم یک به یک جملات را از گوینده دریافت کرده و هر جمله را جداگانه ترجمه کرده و در اختیار شنونده قرار می‌دهد و سپس منتظر جمله



شکل ۱-۱ عملکرد مفسر

بعدی گوینده می‌شود. در این روش برنامه مبدا به زبان ماشین ترجمه نمی‌گردد. در نتیجه در این روش فایل جداگانه‌ای تولید نمی‌گردد. شکل ۱-۱ نحوه عملکرد مفسر را نشان می‌دهد. استفاده از مفسر معایب و مزایایی دارد که در ادامه به آنها خواهیم پرداخت.

مزایای استفاده از مفسر

۱- سهولت اشکال زدایی: در این روش برای شروع اجرای برنامه نیازی به ترجمه کل برنامه به زبان ماشین نیست. زیرا مفسر اولین خط را خوانده و اجرا می‌کند. در نتیجه اجرای برنامه به سرعت شروع می‌شود. این ویژگی برای برنامه در حال ساخت مفید است. زیرا برنامه در حال تغییر است و با هر تغییر نیاز به اجرای مجدد برنامه است. بنابراین، این ویژگی، اشکال‌زدایی را تسریع می‌کند و زمان ساخت نرم‌افزار را کاهش می‌دهد.

۲- قابلیت انعطاف بالا: در این روش، مفسر در زمان اجرای برنامه حضور دارد و برخی اطلاعات که فقط در زمان اجرا قابل دستیابی است توسط مفسر جمع آوری می‌شود و مفسر می‌تواند بر اساس آن تصمیم‌گیری کند.

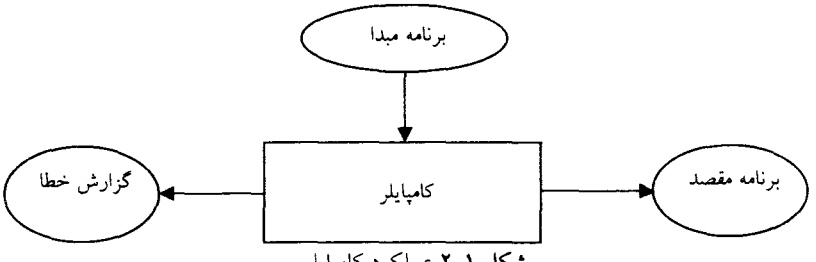
۳- پیاده سازی آسان: پیاده سازی مفسر ساده‌تر از پیاده سازی کامپایلر است.
٤- قابلیت حمل بالا: برنامه توسط مفسر اجرا می‌شود در نتیجه هر جا که مفسر باشد برنامه نیز مستقل از نوع سخت افزار اجرا خواهد شد.

معایب استفاده از مفسر

۱- تکرار تفسیر: در هر اجرا، برنامه مجددا تفسیر می‌شود.
۲- سرعت اجرای پایین: برنامه به طور مستقیم بر روی سخت افزار اجرا نمی‌شود زیرا یک لایه به نام مفسر بین سخت افزار و برنامه قرار می‌گیرد، در نتیجه سرعت اجرای برنامه کاهش می‌یابد.
۳- نیاز به مفسر: در این روش برنامه بدون مفسر اجرا نمی‌شود. در نتیجه هر جا که لازم است برنامه اجرا شود مفسر هم باید وجود داشته باشد در غیر این صورت برنامه قابل اجرا نخواهد بود.
۴- دسترسی به کد منبع[1]: به منظور اجرای برنامه باید کد اصلی منبع موجود باشد تا قابل تفسیر و اجرا باشد. در نتیجه اگر برنامه‌ای به این روش تولید و توزیع گردد، کد منبع آن نیز باید توزیع گردد و توزیع کد برنامه، برای بسیاری از شرکتهای سازنده نرم افزار زیان آور است.

۱-۲-۲ استفاده از کامپایلر

در این روش برنامه بوسیله کامپایلر به زبان ماشین ترجمه می‌شود. کامپایلر نرم‌افزاری است که برنامه نوشته شده به زبان مبدا را به برنامه معادلی در زبان مقصد ترجمه می‌نماید. اگر خطایی در برنامه مبدا وجود داشته باشد کامپایلر آن را گزارش می‌کند. باید توجه کرد که اگر برنامه مبدا صحیح نباشد کامپایلر قادر به ترجمه آن نخواهد بود.



شکل ۱-۲ عملکرد کامپایلر

1. Source Code

عملکرد کامپایلر مانند دوبله یک فیلم خارجی می‌باشد. در این روش ابتدا کل فیلم به زبان فارسی ترجمه می‌شود و سپس در اختیار بینندگان قرار می‌گیرد. در روش استفاده از کامپایلر نیز ابتدا برنامه مبدا بوسیله کامپایلر به زبان ماشین تبدیل می‌شود و سپس این برنامه روی کامپیوتر اجرا می‌شود. استفاده از کامپایلر دارای مزایا ومعایبی است که در ذیل به شرح مختصر هر یک پرداخته می‌شود.

مزایای استفاده از کامپایلر

۱- سرعت اجرای بالا: برنامه مبدا ابتدا به زبان ماشین ترجمه وسپس اجرا می‌شود. برنامه اجرایی بدون واسطه بر روی کامپیوتر اجرا می‌شود در نتیجه سرعت اجرا بالا خواهد بود.

۲- اجرای مستقل برنامه از کامپایلر: بعد از کامپایل، برنامه اجرایی تولید می‌شود. این برنامه اجرایی مستقل از کامپایلر روی ماشین قابل اجرا است و نیازی به وجود کامپایلر در زمان اجرا نیست.

۳- حفاظت از کد منبع برنامه: در این روش برای توزیع برنامه بین کاربران فقط برنامه اجرایی حاصل از کامپایل توزیع می‌گردد و نیازی به توزیع کد منبع نیست، به این ترتیب از کد منبع برنامه حفاظت می‌شود.

٤- عدم تکرار کامپایل: پس از تولید برنامه اجرایی، برنامه روی ماشین‌ها قابل اجرا خواهد بود و نیازی به تکرار کامپایل برای هر اجرا نخواهد بود.

معایب استفاده از کامپایلر

۱- زمانبر بودن اشکال زدایی: در روش کامپایل، ابتدا کل برنامه به زبان مقصد ترجمه و سپس اجرای برنامه شروع می‌شود بنابراین شروع اجرای برنامه قبل از پایان عمل کامپایل امکان پذیر نیست. در نتیجه در مواردیکه برنامه در حال تولید است و تغییرات در برنامه زیاد است و نیاز به اجرای برنامه برای بررسی عملکرد آن زیاد باشد، زمان زیادی صرف کامپایل برنامه خواهد شد.

۲- قابلیت حمل پایین: وقتی برنامه‌ای کامپایل می‌شود برنامه حاصل حاوی دستورات زبان ماشین خاصی خواهد بود که این کد روی انواع ماشینهای دیگر قابل اجرا نخواهد بود. در نتیجه برای هر ماشینی باید جداگانه برنامه کامپایل گردد و برای تولید برنامه اجرایی قابل اجرا روی ماشینهای متفاوت باید از کامپایلر مختلف استفاده نمود.

۳- سهولت پیاده سازی: پیاده سازی کامپایلر از مفسر پیچیده‌تر است.

هر یک از روشهای ذکر شده معایب و مزایای خاص خود را دارند. انتخاب یکی از این دو روش مشکل است. بهترین وضعیت استفاده از هر دو روش است یعنی در زمان تولید از مفسر استفاده شود که در این مرحله از مزایای تفسیر استفاده می‌کنیم و پس از نهایی شدن برنامه، بوسیله کامپایلر برنامه نهایی تولید می‌گردد، تا از مزایای کامپایلر استفاده شود. Visual Basic از هر دو روش پشتیبانی می‌کند. در Visual Basic می‌توان برنامه را به روش تفسیر تولید و اشکال زدایی کرد که در این مرحله از مزایای مفسرها استفاده می‌شود و سرعت تولید بالا خواهد بود و در نهایت، می‌توان برنامه اجرایی را به روش کامپایل تولید کرد. برخی محیط‌های برنامه نویسی نیز فقط یکی از این دو روش را انتخاب می‌کنند. به عنوان مثال زبان C از کامپایل استفاده می‌کند و زبان Lisp از روش تفسیر استفاده می‌کند.

در برخی شبکه‌ها، سیستمهای مختلف سخت افزاری با سیستم‌های عامل متفاوت با یکدیگر ارتباط دارند. یکی از نیازهای عمده در شبکه اجرای یک برنامه در تمام این سیستم‌ها است، این نیاز بوسیله قابلیت حمل برنامه‌ها مرتفع می‌شود. مفسرها دارای قابلیت حمل بالا هستند به همین جهت از مفسرها بسیار استفاده می‌شود. به عنوان مثال صفحات HTML برای نمایش به زبان ماشین ترجمه نمی‌شوند بلکه توسط مرورگرها[1] مانند Internet Explorer و یا Netscape تفسیر می‌شوند. به همین جهت این صفحات در هر سیستم سخت افزاری با هر نوع سیستم عاملی که مرورگر در آنها نصب باشد بدون هیچگونه تغییری قابل نمایش هستند. زبانهای Script مانند VBScript و JavaScript نیز ازجمله زبانهایی هستند که برای اجرا به زبان ماشین ترجمه نمی‌شوند بلکه توسط مرورگرها تفسیر می‌شود. زبان جاوا[2] از امکانات مفسرها و کامپایلرها استفاده می‌کند. زبان جاوا دارای یک کامپایلر است که به جای ترجمه کد جاوا به زبان ماشین، برنامه را به زبان یک ماشین منطقی و مجازی به نام JVM[3] ترجمه می‌کند. JVM یک نرم افزار کوچک است که به سادگی برای هر سخت افزار و سیستم عاملی قابل تهیه است. هر ماشینی که دارای JVM باشد قادر به اجرای برنامه ترجمه شده بوسیله کامپایلر جاوا خواهد بود. نرم افزار JVM برنامه کامپایل شده را به روش تفسیر اجرا می‌کند. زبانهای دیگری مانند C# نیز از روش جاوا پیروی می‌کنند. در زبان C#، CLR[4] نقش ماشین مجازی را بازی می‌کند. مشکل اصلی زبانهایی مانند C# و جاوا سرعت پایین اجرای برنامه است. زیرا یک لایه نرم افزاری که همان ماشین مجازی است بین برنامه و سخت افزار قرار

1. Browser 2. Java 3. Java Virtual Machine
4. Common Language Runtime

دارد. برای حل این مشکل این گونه زبانها به گونه ای طراحی می‌شوند که بتوان برنامه‌ها را به زبان ماشین واقعی نیز کامپایل کرد. برنامه در اولین اجرا به زبان ماشین ترجمه شده در اجراهای بعدی از آن استفاده می‌شود.

از دو روش ذکر شده، به روش ساخت کامپایلر در این کتاب پرداخته خواهد شد. بررسی نحوه ساخت مفسرها خارج از بحث این کتاب است. با در نظر گرفتن تعداد زبانهایی که می‌توانند به عنوان زبان مبدا و زبان به عنوان زبان مقصد به کار روند، تعداد کامپایلرهایی که می‌توان ساخت بسیار زیاد خواهد شد. تکنیکها و روشهایی که برای ساخت کامپایلرها بکار می‌روند مشابه یکدیگرند لذا در این کتاب به صورت کلی به نحوه ساخت کامپایلر پرداخته می‌شود و زبان خاصی مد نظر نیست.

۱-۳ زبان پیاده ساز کامپایلر

کامپایلر برنامه ای است که برنامه به زبان مبدا را به برنامه معادل به زبان مقصد ترجمه می‌کند. کامپایلر نیز یک برنامه است که به یک زبان برنامه نویسی نوشته شده است. زبانی که کامپایلر با آن نوشته می‌شود را زبان پیاده‌ساز[1] می‌نامیم. برای تولید کامپایلر، این برنامه باید کامپایل شود. در این صورت برای تولید کامپایلر نیاز به کامپایلر دیگری می‌باشد. که برای تولید کامپایلر دوم نیز، نیاز به کامپایلر دیگری می‌باشد و این روند ادامه می‌یابد. اولین کامپایلر به زبان اسمبلی نوشته شد و اسمبلر آن به زبان ماشین پیاده سازی گردید.

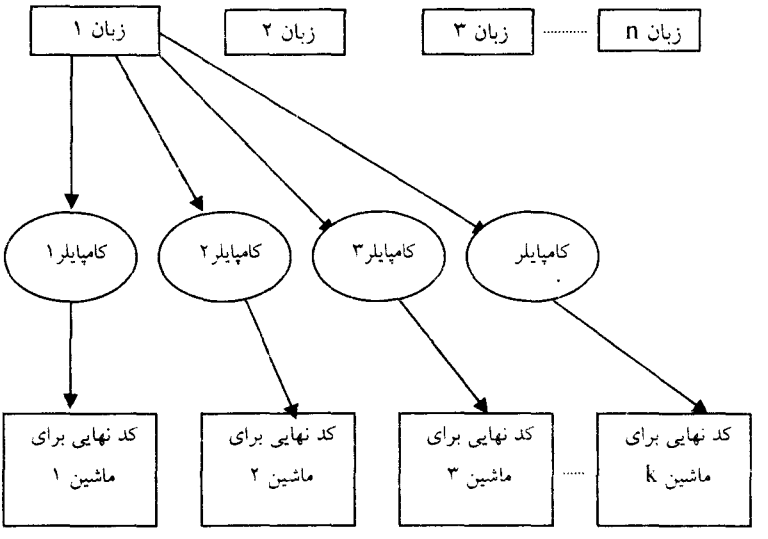
۱-٤ جلوبندی[2] و عقب بندی[3] کامپایلر

می‌خواهیم برنامه‌های به زبان C را روی انواع مختلف کامپیوتر (مانند IBM,MainFram,Sun,Vax) اجرا کنیم در این صورت برای هر نوع کامپیوتر، باید یک کامپایلر جداگانه بسازیم. اگر n تعداد زبانهای برنامه سازی (مانند C و پاسکال و ...) و k تعداد انواع مختلف کامپیوترها باشد در این صورت به nk کامپایلر نیاز است. اگر n=۱۰ و k=۲۰ باشد در این صورت به nk=۲۰۰ کامپایلر مختلف نیاز است. ایجاد این تعداد کامپایلر بسیار زمانبر و پر هزینه است (شکل ۱-۳). برای حل این مشکل از تقسیم کامپایلر به جلوبندی و عقب بندی استفاده می‌کنیم. برای تشریح جلوبندی و عقب بندی کامپایلر از یک مثال استفاده می‌کنیم. فرض کنیم می‌خواهیم متن خبری را از زبان روسی به تمام زبانهای دنیا ترجمه کنیم

---

1. Implementation Language 2. Front End 3. Back End

واضح است که اگر تعداد کل زبانهای دنیا را n فرض کنیم در این صورت به n-1 (به جز زبان روسی) مترجم نیاز است تا این خبر را ترجمه کنند.



شکل۱-۳ نحوه کامپایل یک برنامه برای اجرا روی کامپیوترهای مختلف

اگر بخواهیم متن هر خبری را از هر زبانی به هر زبان دیگر ترجمه کنیم در این صورت به n(n-1)/2 مترجم نیاز است[1]. در این صورت اگر n=۱۰۰ باشد تعداد مترجمهای مورد نیاز به صورت زیر محاسبه می‌شود.

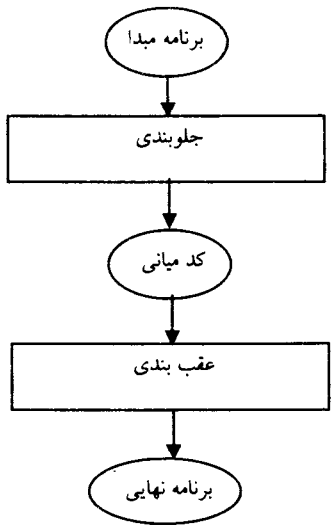
(۱۰۰(۱۰۰-۱))/۲=۴۹۵۰

راه حل این مشکل استفاده از یک زبان میانی است. به عنوان مثال زبان انگلیسی را به عنوان زبان میانی و بین المللی در نظر می‌گیریم هر متنی ابتدا به زبان انگلیسی ترجمه شده و سپس به زبان مقصد ترجمه می‌شود. در این صورت متن خبر از زبان مبدا به زبان انگلیسی و سپس از زبان انگلیسی به زبان مقصد ترجمه می‌شود. در این حالت فقط به n مترجم احتیاج است. اگر n=۱۰۰ باشد فقط به ۱۰۰ مترجم احتیاج است که بسیار کمتر از ۴۹۵۰ است.

---

۱. زیرا هر خبری از زبان مبدا به n-1 زبان دیگر ترجمه می شود و با توجه به اینکه n زبان وجود دارد بنابراین n(n-1) مترجم نیاز است. با توجه به اینکه هر مترجمی می تواند هر خبری را از زبان مبدا به زبان مقصد و بالعکس ترجمه کند در نتیجه به n(n-1)/2 مترجم نیاز است.

در زبانهای برنامه نویسی مانند مسئله ترجمه متون می‌توان از زبان میانی استفاده کرد[1]. یک زبان میانی در نظر می‌گیریم. در ابتدا برنامه مبدا را به این زبان میانی ترجمه کرده سپس از زبان میانی به زبان مقصد ترجمه می‌کنیم. بخشی از کامپایلر که وظیفه ترجمه برنامه مبدا به برنامه زبان میانی را بر عهده دارد، جلوبندی و بخشی از کامپایلر که وظیفه ترجمه برنامه به زبان میانی را به زبان مقصد بر عهده دارد را عقب بندی کامپایلر می‌نامیم(شکل ۱-٤).



شکل ۱-٤ جلوبندی و عقب بندی

با استفاده از این روش برای n زبان مبدا و k کامپیوتر مختلف به n جلوبندی و k عقب بندی نیاز است که در مجموع n+k برنامه احتیاج است (شکل ۱-۵). جلوبندی کامپایلر به زبان مبدا و عقب بندی کامپایلر به زبان مقصد وابسته است.

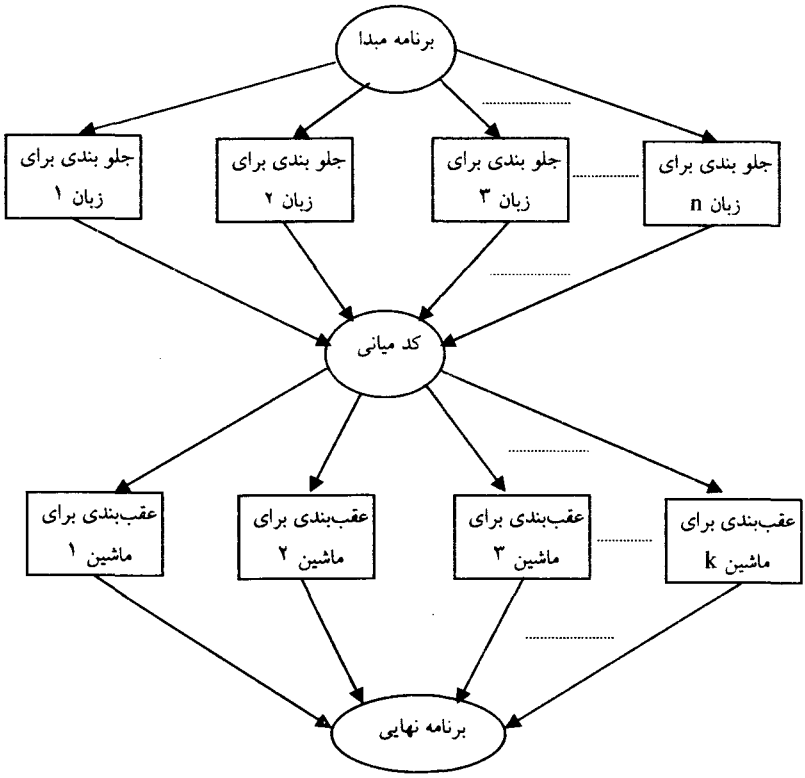
تقسیم کامپایلر به عقب بندی و جلوبندی مزایایی دارد که مهمترین آنها عبارتند از:

- سادگی طراحی
- استقلال جلوبندی از زبان مقصد
- استقلال عقب بندی از زبان مبدا
- کاهش پیچیدگی

---

۱. مثال ترجمه خبر با ترجمه یک برنامه، فرقهایی دارد به عنوان نمونه یک مترجم قادر به ترجمه زبان A با B و بالعکس می باشد، در حالیکه کامپایلر برنامه را از زبان مبدا به زبان مقصد ترجمه می کند و قادر به انجام عمل عکس نیست.

- افزایش قابلیت استفاده مجدد
- افزایش سرعت تولید کامپایلر برای سخت افزار جدید و زبانهای جدید

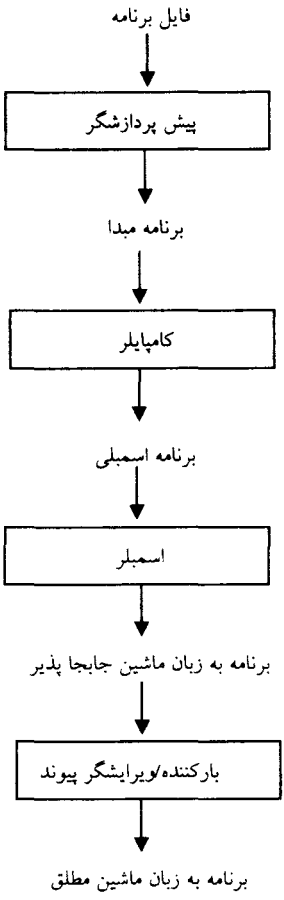


شکل ۱-۵ استفاده از زبان میانی برای تولید کامپایلرهای مختلف

۱-۵ ساختار محیطهای برنامه‌نویسی

برای نوشتن برنامه به یک زبان خاص و تبدیل آن به برنامه اجرایی، اجزای مختلفی در کنار کامپایلر قرار می‌گیرند که هر یک وظیفه‌ای را بر عهده دارند. امروزه محیطهای برنامه نویسی تمام امکانات لازم برای نوشتن برنامه، خطایابی و تولید کد نهایی را در اختیار برنامه نویس قرار می‌دهند، این گونه محیطها را IDE[1] می‌نامیم. شکل ۱-۶ اجزای اصلی یک محیط IDE را نشان می‌دهد.

1. Integrated Development Environment



شکل ۱-۶ جایگاه کامپایلر در IDE

۱-۵-۱ پیش پردازنده

پیش‌پردازنده وظیفه انجام برخی تبدیلات اولیه را بر روی برنامه ورودی بر عهده دارد. در واقع پیش پردازنده یک کامپایلر سطح بالا است که برخی ترجمه‌ها را انجام می‌دهد. از مهمترین وظایف پیش‌پردازشگر می‌توان به موارد ذیل اشاره کرد:

الف- پردازش ماکروها:

این بخش از پیش پردازشگر به برنامه‌نویسان امکان تعریف و استفاده از ماکروها را جهت تسهیل و تسریع برنامه‌نویسی می‌دهد. به عنوان مثال در زبان C++ می‌توان برنامه‌ای را به صورت ذیل نوشت:

```
#define MAX 20
int main(){
int i;
for(i=0;i<MAX;i++)
      cout<<i;
}
```

پیش‌پردازنده این قطعه برنامه را به برنامه ذیل تبدیل می‌کند.

```
int main(){
int i;
for(i=0;i<20;i++)
      cout<<i;
}
```

همانطور که ملاحظه می‌شود پیش‌پردازشگر برنامه ورودی را به برنامه ای تبدیل می‌کند که به جای MAX عدد 20 قرار گرفته است. به عنوان مثال برنامه ذیل را در نظر بگیریم.

```
#define line cout<<"**************************"
int main(){
line;
cout<<"Hello To World";
line;
}
```

پیش پردازشگر برنامه را دریافت کرد وخروجی ذیل را ایجاد می‌کند .

```
int main(){
cout<<"**************************";
cout<<"Hello To World";
cout<<"**************************";
}
```

ب- الحاق فایلها:

برنامه‌ها معمولا از توابع کتابخانه ای استفاده می‌کنند این توابع در فایلهایی قرار دارند که الحاق و پیوند این فایلها به برنامه وظیفه پیش‌پردازشگر است. به عنوان مثال برنامه ذیل را در نظر گرفته می‌شود. به عنوان مثال دیگر برنامه ذیل را در نظر بگیریم.

```
#include <iostream.h>
int main(){
cout<<"Welcome";
}
```

این برنامه توسط پیش‌پردازنده پردازش می‌شود، خروجی پیش‌پردازشگر برنامه ای است که علاوه بر متن برنامه اصلی محتوای iostream به آن اضافه گردیده است.

ج-تعمیم زبان:
پیش پردازنده‌ها می‌توانند قابلیتهایی را به زبان اضافه کنند. به عنوان مثال برای دسترسی به پایگاه داده در زبان ماکروهایی وجود دارد که توسط پیش پردازنده‌ها پردازش می‌شوند.

۱-۵-۲ کامپایلر
کامپایلر برنامه خروجی پیش پردازنده را به زبان اسمبلی ترجمه می‌کند. این بخش بیشترین وظایف در تبدیل برنامه مبدا به یک برنامه اجرایی را بر عهده دارد.

۱-۵-۳ اسمبلر
این بخش برنامه اسمبلی خروجی از بخش کامپایلر را به کدهای باینری (صفرویک)که قابل فهم برای کامپیوتر باشد تبدیل می‌کند.

۱-۵-٤ بار کننده ویرایشگر پیوند
وظیفه این بخش انتقال برنامه اجرایی به حافظه به منظور اجرا می‌باشد.

۱-٦ فازهای کامپایلر
همانطور که قبلا ذکر شد کامپایلر برنامه به زبان مبدا را به زبان مقصد ترجمه می‌کند. در نتیجه با توجه به تعداد زبانهای مبدا و زبانهای مقصد کامپایلرهای بسیاری می‌توان ایجاد کرد. اما ساختار و اصول همه این کامپایلرها یکسان است. در این بخش ساختار کلی کامپایلر را بررسی می‌کنیم.

فرآیند کامپایل بسیار پیچیده است، در نتیجه برای ساخت چنین نرم افزاری، آن را به بخشهای مختلف تقسیم می‌کنند به طوری که هر بخشی وظیفه مشخصی بر عهده دارد و ترکیب وهمکاری این بخشها با یکدیگر کامپایلر را می‌سازد. هر یک از این بخشها، یک فاز نامیده می‌شود. هر فاز کامپایلر وظیفه به خصوصی بر عهده دارد. تقسیم کامپایلر به فازهای مختلف مزایایی دارد که عبارتند از :

۱- سادگی طراحی
۲- افزایش کارایی
۳- افزایش قابلیت حمل
٤- افزایش قابلیت انعطاف پذیری

شکل ۱-۷ فازهای کامپایلر را نشان می‌دهد. هر فاز، خروجی فاز قبلی را دریافت و نمایش دیگری را ایجاد کرده و تحویل فاز بعدی می‌دهد. لازم به ذکر است که هر یک از این فازها الزاما به صورت یک برنامه نرم افزاری جدا پیاده سازی نمی‌شوند بلکه ممکن است چند فاز در قالب یک برنامه پیاده سازی شوند. در نتیجه ممکن است نیازی به ایجاد صریح نمایشهای میانی نباشد. در ادامه به تشریح مختصر هر یک از این فازها خواهیم پرداخت.

۱-۶-۱ تحلیلگر لغوی[1]

تحلیلگر لغوی بخشی از کامپایلر است که مستقیما به برنامه مبدا دسترسی دارد. تحلیلگر لغوی برنامه مبدا را به صورت جریانی از کاراکترها دریافت کرده لغات تشکیل دهنده برنامه و نوع آنها را تشخیص داده و برای تحلیلگر نحوی ارسال می‌کند. به عنوان مثال قطعه برنامه ذیل را در نظر می‌گیریم.

```
K:= H+12*B;
```

تحلیلگر لغوی با دریافت این قطعه برنامه اطلاعات ذیل را به تحلیلگر نحوی ارسال می‌کند.

شناسه K

علامت انتساب :=

شناسه H

علامت +

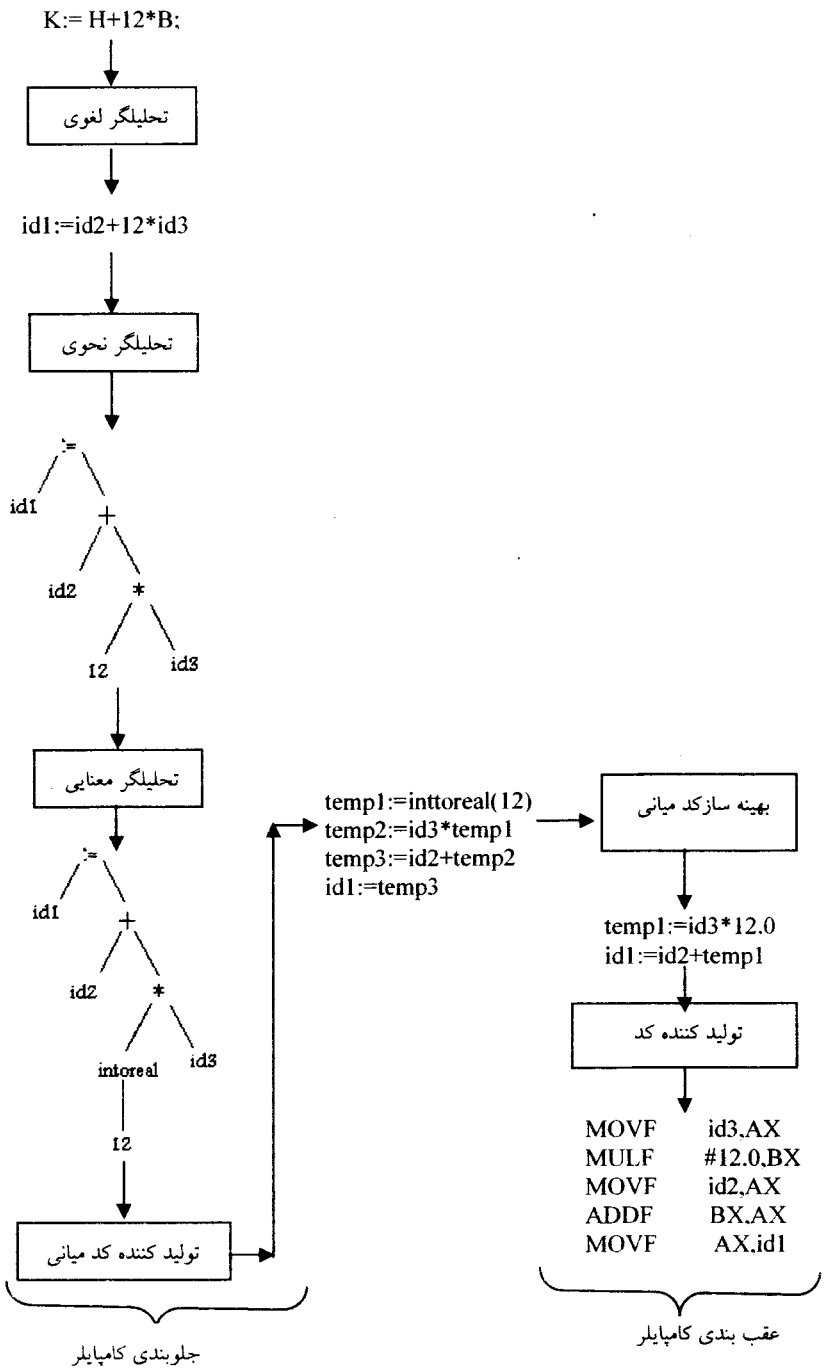
عدد 12

علامت *

شناسه B

علامت ;

1. Lexical Analayzer



شکل ۱-۷ فازهای کامپایلر

کشف خطاهای مربوط به ساختار تک تک لغات نیز وظیفه تحلیلگر لغوی است. به عنوان مثال اگر در زبان پاسکال یک شناسه به صورت ذیل تعریف شود. تحلیلگر لغوی این لغت را به عنوان خطا گزارش می‌کند. زیرا طبق دستور زبان پاسکال یک شناسه نباید با عدد شروع شود.

```
var 7temp:integer;
```

هر یک از عبارات ذیل دارای خطای لغوی در زبان پاسکال می‌باشند.

```
Temp#:=10;
t := 123.423.34;
str := 'ali;
```

برای روشن شدن هر چه بیشتر کارکرد تحلیلگر لغوی از یک مثال ساده استفاده می‌کنیم. فرض کنید جمله ذیل را می‌خواهیم از زبان انگلیسی به زبان فارسی ترجمه کنیم.

```
@aq )ewr %tywqe56 ewiu23.
```

واضح است که هیچ مترجمی قادر به ترجمه جمله فوق نیست زیرا بررسی تک تک لغات نشان می‌دهد که هیچ یک از این دنباله کاراکترها، یک لغت معتبر در زبان انگلیسی نیست.

۱-۶-۲ تحلیلگر نحوی[1]

وظیفه تحلیلگر نحوی بررسی صحت و درستی ترتیب لغات برنامه مبدا است. برای روشن شدن هر چه بیشتر کارکرد تحلیلگر نحوی از مثال ساده ذیل استفاده می‌کنیم. فرض کنید جمله ذیل را می‌خواهیم از زبان انگلیسی به زبان فارسی ترجمه کنیم.

```
are am is I you is.
```

مترجم با بررسی این جمله، ٦ لغت را تشخیص می‌دهد. تک تک لغات این جمله در زبان انگلیسی معتبر هستند. این وظیفه را تحلیلگر لغوی در کامپایلر انجام می‌دهد. علی رغم تایید صحت تک تک لغات هنوز قادر به ترجمه این جمله نیستیم. زیرا ترتیب قرار گرفتن این لغات در جمله از نظر دستور زبان انگلیسی مجاز نیست. واضح است که هیچ مترجمی قادر به ترجمه جمله فوق نیست. یک جمله ساده در زبان انگیسی دارای ساختار ذیل است.

```
.مفعول فعل فاعل
```

جمله فوق این ساختار و هیچ ساختار دستوری دیگر زبان انگلیسی را رعایت نکرده است. بررسی ساختار برنامه مبدا از وظائف تحلیلگر نحوی است. تحلیلگر نحوی خروجی تحلیلگر لغوی را دریافت کرده و اگر ترتیب لغات برنامه مبدا صحیح باشد از آن یک ساختار درختی تولید می‌کند. این درخت، نشان دهنده ساختار برنامه مبدا است که اصطلاحا به آن درخت تجزیه گفته می‌شود(درخت تجزیه در فصلهای آینده تشریح خواهد شد.). کشف

---

1. Syntax Analayzer

خطاهای نحوی که مربوط به ترتیب لغات برنامه مبدا می‌باشد از وظائف تحلیلگر نحوی است. به عنوان مثال در زبان پاسکال هر یک از عبارات ذیل دارای خطای نحوی می‌باشد.

```
1- A:=>B
2- A B:=;
3- if (a =b  then
4- program a;
        begin
          writeln('hello');
        ed
```

عدم رعایت ترتیب صحیح در عبارت انتساب، عدم توازن پرانتزها، عدم درج end در آخر برنامه از جمله خطاهای نحوی این عبارات است که کشف آن وظیفه تحلیلگر نحوی می‌باشد. دقت کنید که عبارات بالا همگی از نظر تحلیلگر لغوی صحیح بوده و دارای هیچ خطایی نیستند.

۱-۶-۳ تحلیلگر معنایی[1]

برای بیان وظائف این قسمت از کامپایلر از همان مثال ترجمه یک جمله استفاده می‌کنیم. سوالی که مطرح است این است که آیا هر جمله انگلیسی که از لحاظ لغوی و دستوری صحیح باشد آیا قابل ترجمه است. برای پاسخ به این سوال جمله ذیل را در نظر می‌گیریم.

```
I  run water
```

علی رغم این که این جمله از نظر لغوی صحیح است (یعنی تک تک لغات در زبان انگلیسی مجاز است) و از نظر نحوی نیز صحیح است ولی این جمله قابل ترجمه نیست (اگر این جمله را ترجمه کنیم، ترجمه آن جمله " من آب را دویدم " می‌شود که بی معنی است).

وقتی تحلیل لغوی و تحلیل نحوی برنامه مبدا به پایان رسید، هنوز مواردی باقی مانده است که نیاز به بررسی دارند. تحلیلگر معنایی مواردی که در دو مرحله قبلی بررسی نشده است را بررسی می‌کند. تحلیلگر معنایی معنی دار بودن عباراتی که از نظر نحوی درست بوده‌اند را مورد بررسی قرار می‌دهد.

تحلیلگر معنایی وظیفه تشخیص خطاهای معنایی، جمع آوری اطلاعات لازم برای مراحل بعد کامپایلر و اعمال برخی تغییرات برای اصلاح برخی موارد را بر عهده دارد. برخی از مواردی که توسط تحلیلگر معنایی انجام می‌شود عبارتند از:

---

1. Semantic Analayzer

- بررسی هماهنگی پارامترها:

فراخوانی یک روال باید با تعریف روال هماهنگی داشته باشد. به عنوان مثال در زبان C تعداد پارامترهای ارسال نوع آنها و ترتیب آنها باید با تعریف تابع هماهنگی داشته باشد. به عنوان مثال برنامه (به زبان C) ذیل را در نظر می‌گیریم.

```
int sum(int a, int b){
return a+b;
}
int main(){
int b=12,c=13;
float  i=34;
b=sum(i,b,c);
return 0;
}
```

با دقت در این برنامه در می‌یابیم که این برنامه از لحاظ لغوی (تمام لغات به کار رفته در زبان C مجاز هستند.) و نحوی (ترتیب لغات بکار رفته در هر عبارت به تنهایی صحیح است.)صحیح است، اما دارای اشکالات معنایی است. زیرا تابع sum با سه آرگومان فراخوانی شده است، در صورتیکه طبق تعریف تابع sum این تابع فقط دو پارامتر را می‌پذیرد. همچنین متغیر i از نوع float می‌باشد در حالیکه تابع sum آرگومان از نوع float را نمی‌پذیرد.

- بررسی و کنترل نوع[1]:

در این قسمت نوع عملوندهای یک عملگر مورد بررسی قرار می‌گیرد. به عنوان مثال به قطعه برنامه ذیل که به زبان پاسکال نوشته شده است توجه کنید.

```
var   f:real;
      a:array [1..10] of integer;
begin
f:=2;
a[f]:=12;
end.
```

این برنامه دارای خطا است، زیرا اندیس آرایه نباید عدد اعشاری باشد. کشف این خطا بر عهده تحلیلگر معنایی است. البته این مورد در زبانهای مختلف ممکن است متفاوت باشد. به عنوان مثال برنامه ذیل در زبان Borland C++ 3.1 بدون خطا است.

```
float  a;
int    b[10];
a=1.2;
b[a]=12;
cout<<b[1];
```

در این قطعه برنامه در عبارت b[a]، a عدد 1 (که صحیح است) در نظر گرفته می‌شود و در نتیجه b[1]=12 خواهد شد و خطایی گزارش نمی‌شود.

---

1. Type Checking

همانطور که ملاحظه گردید کامپایلر زبان پاسکال از اندیس اعشاری برای آرایه خطا گرفت در صورتیکه کامپایلر C خطایی گزارش نکرد. تفاوت این دو کامپایلر در قوانین معنایی آنها و نحوه عملکرد تحلیلگر معنایی آنها است.

- تبدیل نوع

قطعه برنامه ذیل را در نظر بگیرید:

```
int a=10;
float  b=12.3;
float  c;
c=a+b;
```

عملیات جمع برای اعداد صحیح واعشاری متفاوت است برای انجام عمل جمع بالا باید عملوندهای، عملگر از یک نوع باشند، به همین جهت تحلیلگر معنایی عمل ارتقا نوع را انجام می‌دهد بدین معنا که تحلیلگر معنایی متغیر a را از نوع int موقتا به نوع float تبدیل می‌کند تا عمل جمع قابل انجام باشد.

- بررسی تعریف دوباره متغیر

تحلیلگر معنایی بررسی می‌کند تا هیچ متغیری دو بار تعریف نشده باشد.

۱-٦-٤- تولید کننده کد میانی[1]

پس از بررسیها وتبدیلات انجام شده در مراحل تحلیل لغوی، نحوی، معنایی و اطمینان از صحت برنامه مبدا، تولید کد میانی آغاز می‌گردد. در این مرحله بخش تولید کننده کد میانی از خروجی تحلیلگر معنایی کد میانی را تولید می‌کند. کد میانی، زبان ماشین یک ماشین منطقی و مجازی می‌باشد. به همین جهت هر کامپایلر می‌تواند کد میانی خاص خود را تعریف کند. این کد باید حداقل خواص ذیل را دارا باشد.

- **سهولت تولید کد:** زبان ماشین منطقی تا حد امکان باید ساده در نظر گرفته شود تا تولید کد برای آن آسان باشد.
- **سهولت ترجمه به زبان مقصد:** زبان ماشین منطقی باید به گونه‌ای باشد که تبدیل آن به زبان ماشین آسان باشد.

یکی از انواع زبانهای منطقی،کد سه آدرسه نامیده می‌شود. این زبان برای ماشینی است که تمام حافظه آن رجیستر است یعنی فرض می‌شود این ماشین به تعداد مورد نیاز رجیستر برای پردازش اطلاعات دارا می‌باشد. در کد سه آدرسه تمام دستورات حاوی عملگر انتساب

1. Intermediate Code (IC)

هستند. بجز عملگر انتساب حداکثر یک عملگر دیگر مجاز است. به عنوان مثال عبارت ذیل چند دستورالعمل به زبان کد سه آدرسه است.

```
temp1:=inttoreal(12)
temp2:= id3*temp1
temp3:=id2+temp2
id1:=temp3
```

۱-۶-۵ بهینه کننده کد میانی

بهینه کننده کد وظیفه بررسی کد میانی را بر عهده دارد تا در صورت امکان آن را بهینه سازی کند. بهینه سازی یعنی اعمال تغییراتی در برنامه که بدون تغییر در عملکرد برنامه، مصرف حافظه کاهش و یا سرعت اجرای برنامه، افزایش یابد. به عنوان مثال عبارت زیر را در نظر می‌گیریم.

```
temp1:=id3*inttoreal(12);
```

در این عبارت، در زمان اجرا عدد 12 به 12.0 تبدیل می‌شود سپس در عمل جمع شرکت می‌کند. این عبارت را می‌توان به صورت ذیل تبدیل نمود.

```
temp1:=id3*12.0;
```

در این عبارت نیازی به تبدیل 12 به 12.0 در زمان اجرا نمی‌باشد، در نتیجه بدون اعمال تغییر در عملکرد برنامه سرعت اجرای آن را افزایش داده‌ایم. به عنوان مثال دیگر دستورات ذیل را در نظر می‌گیریم.

```
temp1:=inttoreal(12)
temp2:= id3*temp1
temp3:=id2+temp2
id1:=temp3
```

این دستورات را می‌توان به صورت ذیل بهینه کرد.

```
temp1:=id3*12.0
id1:=id2+temp1
```

همانطور که ملاحظه می‌شود این تغییر نیز باعث حذف دو عمل انتساب (افزایش سرعت) و حذف متغیرهای temp2 و temp3 (کاهش مصرف حافظه) شده است.

۱-۶-۶ تولید کننده کد

بخش تولید کننده کد، کد میانی بهینه سازی شده را به زبان اسمبلی ترجمه می‌کند، به عنوان مثال دستورات قبل به صورت ذیل ترجمه می‌شوند.

```
MOV     ID3,AX
MULF    #12.0,BX
MOVF    ID2,AX
ADDF    BX,AX
MOVF    AX,ID1
```

این برنامه توسط اسمبلر به زبان ماشین ترجمه می‌شود.

به جز موارد نشان داده شده در شکل ۱-۷ ، اجزای دیگری نیز در کامپایلر وجود دارد. جدول نماد یکی از بخشهایی است که با همه فازها در ارتباط است. جدول نماد مکانی است که اطلاعات مربوط به اسامی را نگهداری می‌کند. فازهای مختلف ممکن است اطلاعاتی به جدول نماد اضافه و یا از اطلاعات جدول نماد برای انجام وظیفه خود استفاده کند. جدول نماد در بخشهای بعدی معرفی خواهد شد.

یکی دیگر از بخشهای کامپایلر، اداره کننده خطا است. این بخش وظیفه مدیریت خطاهایی را دارد که ممکن است فازهای مختلف به آنها برخورد کنند.

نرم افزارهای دیگر نیز ممکن است از ساختار مشابه کامپایلر پیروی کنند. به عنوان مثال مفسرها فاز تحلیل لغوی ، نحوی ، معنایی و تولید کد میانی را انجام می‌دهند. اما پس از تولید کد میانی به جای ترجمه آن به زبان ماشین، کد میانی را اجرا می‌کنند. همچنین نرم افزارهایی که دستورات SQL را اجرا می‌کنند فاز تحلیل لغوی و نحوی را انجام می‌دهند.

۱-۷ مدل تجزیه[1] و ترکیب[2]

فرآیند کامپایل شامل دو مرحله است. در مرحله اول برنامه به اجزای تشکیل دهنده آن تجزیه می‌شود. در طی این مرحله خطاهای برنامه مبدا مشخص می‌گردد. فازهای تحلیل لغوی، تحلیل نحوی و تحلیل معنایی و تولید کد میانی عملیات تجزیه را انجام می‌دهند این فازها، جلوبندی کامپایلر را تشکیل می‌دهند. پس از اطمینان از صحت برنامه مبدا برنامه تجزیه شده برای ایجاد برنامه مقصد ترکیب می‌گردد. فازهای بعدی عملیات ترکیب را انجام می‌دهند. بقیه فازها در عقب بندی کامپایلر قرار دارد.

۱-۸ ویژگیهای کامپایلر خوب

برای یک زبان برنامه‌سازی می‌توان کامپایلرهای مختلفی تولید کرد اما سوال این است که کدام کامپایلر بهتر است و یا ویژگیهای یک کامپایلر خوب چیست؟ برای یک کامپایلر ویژگیهای مختلفی می‌توان در نظر گرفت که مهمترین آنها عبارتند از:

۱- تولید کد صحیح: کامپایلر باید بتواند برای برنامه مبدا کد صحیح تولید کند، کامپایلری که حتی یک کد اشتباه تولید کند قابل استفاده نیست.

---

1. Analytic Phase
2. Synthetic Phase

۲- پیروی از مشخصات زبان مبدا: کامپایلر باید تمام مشخصات زبان مبدا اعم از لغوی، نحوی و دیگر مشخصات تعریف شده زبان را پشتیبانی کند.

۳- امکان پردازش برنامه‌های بزرگ: کامپایلر باید بتواند برنامه‌های حجیم را پردازش کرده و کد لازم را تولید کند. البته برخی محدودیتهای دیگر مانند کمبود حافظه اصلی می‌تواند روی این قابلیت تاثیر بگذارد.

به عنوان مثال در زبانهای برنامه‌سازی ساختار زبان به صورت منطقی بیان می‌شوند به عنوان مثال ساختار case در پاسکال به صورت زیر تعریف می‌شود.

Case متغیر of

: مقدار اول

:مقدار دوم

END;

کامپایلری خوب است که برای تعداد حالات (تعداد مقادیر) دستور Case محدودیت نداشته باشد به عنوان مثال آیا می‌توان یک دستور Case با ۱۰۰۰ حالت مختلف نوشت آیا کامپایلر می‌تواند چنین دستوری را پردازش کند. یا آیا می‌توان برای روالی ۱۰۰۰ متغیر محلی مختلف تعریف کرد.

کامپایلرها برای بسیاری از بخشهای زبان مبدا حافظه محدودی در نظر می‌گیرد. به عنوان مثال بسیاری از کامپایلرها فرض می‌کنند که در روالها بیشتر از ۳۲ متغیر تعریف نمی‌شود. در حالیکه برنامه‌هایی وجود دارند که برنامه تولید می‌کنند[1] این گونه برنامه‌ها ممکن است برنامه‌هایی تولید کنند که از یک واحد زبان (مثلا دستور case ی تولید کنند که تعداد حالات زیادی داشته باشد.) نمونه بزرگی تولید کنند.

٤- سرعت کامپایلر: هر چه سرعت کامپایلر بیشتر باشد سرعت تولید نرم افزار کمتر و در نتیجه هزینه تولید نرم افزار کمتر می‌شود.

٥- اندازه کامپایلر: هر چه اندازه کامپایلر کوچکتر باشد روی حافظه کمتری قابل اجرا است البته با توجه کاهش قیمت حافظه‌ها و افزایش حافظه کامپیوترها از اهمیت این ویژگی کاسته شده است اما در کاربردهای خاص مثل تعبیه یک کامپایلر در یک وسیله الکترونیکی مثل تلفن، اندازه کامپایلر خیلی مهم است.

---

۱. این گونه برنامه را Code Generator می نامند.

٦- پیغامهای خطای کامپایلر: پیغامهای خطای تولید شده توسط کامپایلر باید واضح باشد. اگر گزارش خطا، شامل علت خطا، موقعیت وقوع خطا، نام فایلی که خطا در آن رخ داده است و یا توصیه‌ای جهت رفع خطا باشد، رفع خطا با سرعت بیشتری انجام می‌شود. البته بیان علت خطا برای کامپایلر بسیار مشکل است.

۷- قابلیت حمل: برنامه‌ای قابل حمل است که برای اجرا شدن در ماشینهای مختلف به تغییرات زیادی احتیاج نداشته باشد. در مورد کامپایلر دو نوع قابلیت حمل مد نظر است اولی قابلیت حمل کامپایلر است (یعنی خود کامپایلر در کامپیوترهای مختلف اجرا شود.). اگر کامپایلر به یکی از زبانهای برنامه‌نویسی سطح بالا نوشته شده باشد قابلیت حمل آن بالا خواهد بود. دوم قابلیت حمل کد تولید شده است که قابلیت هدف‌گیری مجدد[1] نامیده می‌شود. برای حصول این ویژگی عقب بندی کامپایلر تحت تاثیر قرار می‌گیرند. البته تمام قسمتهای عقب بندی تغییر نمی‌کند فقط آن بخشی که کدهایی را تولید می‌کند تغییر می‌کند.

۸- بهینه‌سازی: بهینه‌سازی یکی از کارهای جنبی کامپایلر محسوب می‌شود. کامپایلری که بتواند کد بهینه تری تولید کند مطلوبتر است. بهینه‌سازی آخرین مرحله از کامپایلر می‌باشد. کشف روشها و تکنیکهای بهینه‌سازی یکی از زمینه‌های تحقیقاتی مهم در کامپایلرها می‌باشد.

۱-۹ زمینه‌های تحقیقاتی کامپایلر

اصطلاح کامپایلر اولین بار توسط شخصی به نام هوپر[2] در سال ۱۹۵۰ میلادی ابداع گردید. اولین کامپایلر زبانهای سطح بالا برای زبان Fortran در طی سالهای ۱۹۵٤ تا ۱۹۵۷ توسط گروهی به سرپرستی جان باکاس[2] در IBM و در ۱۸ نفر-سال[3] تولید شد.

طی سالهای ۱۹٦۰ تا ۱۹۷۱ زبانهای برنامه‌سازی جدیدی به وجود آمدند در نتیجه سرعت تولید کامپایلر بیشتر از کیفیت کامپایلر اهمیت داشت. در این دوران متخصصین بیشتر روی جلوبندی کامپایلر تحقیق کردند که منجر به ایجاد تکنیکهای قوی در ساخت جلوبندی شد.

از سال ۱۹۷۵ ایجاد زبانهای جدید کاهش یافت در نتیجه سرعت تولید کامپایلر از اهمیت کمتری برخوردار شد و کیفیت تولید کامپایلر بیشتر مورد توجه قرار گرفت. مواردی از قبیل: برنامه نویسی تابعی و توزیعی، فراخوانی روال از راه دور، بهینه سازی کد، تخصیص و آزاد سازی حافظه در زمان اجرا مورد توجه قرار گرفت.

---

1. Retargetaibility
1. Grace Murray Hopper
2. John Backus

تمرینات

۱. تفاوتهای کامپایلر و مفسر را بیان کنید.

۲. جلوبندی و عقب‌بندی درکامپایلر را شرح دهید؟ مزایای تقسیم‌بندی به جلوبندی و عقب‌بندی را شرح دهید؟

۳. فازهای مختلف کامپایلر را نام برده و هر یک را تشریح کنید؟

٤. هر کدام از عبارات ذیل که به زبان پاسکال است چند لغت دارد.

الف- w12 := '125';

ب-
```
while i<20 do
  begin
  i:=i+1;
  a=a*a;
  end
```

۵. عبارت ذیل را به کد سه آدرسه تبدیل کنید.

```
a=b*2+d*2;
b=1;
d=2*b+b+b;
```

٦. کد سه آدرسه حاصل از تمرین ۵ را بهینه کنید.

۷. کد ذیل را به کد سه آدرسه تبدیل کرده و نتیجه را بهینه نمایید.

```
a=21+(b+c);
d=2*b+2*c;
c=42+2*b+2*c;
```

۸ کد ذیل را به کد سه آدرسه تبدیل کرده و نتیجه را تا حد امکان بهینه نمایید.

```
i=0;
while(i<10)
{
d=2*b+2*c;
c=42+2*b+2*c
i++;
}
```

۹. چه خطاهای معنایی ممکن است در عبارت ذیل که به زبانc نوشته شده است رخ دهد؟

```
C=B/A;
```

۱۰. هدف از این تمرین ساخت یک مفسر ساده برای یک زبان است. زبان ساده ای را با ساختار ذیل در نظر بگیرید. در این عبارات operator یکی از عملگرهای - , * , / , + و operand یکی از شناسه‌های A,B,C,D است و num یک عدد صحیح است.

```
operand = num ;
operand = operand   operator   operand ;
```

برنامه ای بنویسید که عبارات با ساختار فوق را بخواند عملیاتها را انجام داده و در نهایت محتوای A را چاپ کند. به عنوان مثال اگر ورودی عبارات ذیل باشد برنامه عدد ۱۰ را چاپ می‌کند.[۱]

```
A = 2
B =3
C= A+B
B=2
A= B * C
```

۱۱. هدف از این تمرین ساخت یک کامپایلر ساده برای یک زبان جدید است. برای تولید کامپایلر یک زبان جدید می‌توان از امکانات زبانهای موجود بهره برداری کرد. به این منظور ابتدا زبان جدید را به یکی از زبانهای موجود ترجمه کرده و با استفاده از کامپایلر زبان موجود، برنامه را به زبان ماشین و قابل اجرا ترجمه می‌کنیم.

زبان ساده ای را با ساختار ذیل در نظر بگیرید. در این عبارات، operator یکی از عملگرهای - , * , / , + و operand یکی از شناسه‌های A,B,C,D است و num یک عدد صحیح است.

```
operand = num ;
operand = operand  operator  operand;
```

برنامه ای بنویسید که عبارات با ساختار فوق را بخواند سپس برنامه ای به زبان C++ تولید کند که همین عملیات را انجام دهد و در نهایت محتوای A را چاپ کند. در واقع برنامه مورد نظر باید این عبارات را به زبان C++ ترجمه کند. به عنوان مثال اگرورودی، عبارات ذیل باشند.

```
A = 2
B =3
C= A+B
B=2
A= B * C
```

برنامه‌ای به زبان C++ به صورت ذیل تولید می‌شود.[۲]

```
#include<iostream.h>
int main()
{
int A,B,C,D;
A=2;
B=3;
C=A + B;
B=2;
A= B * C;
cout<<A;
return 0;
}
```

---

۱. این برنامه در واقع یک مفسر است و به روش مفسرها باید عمل کند. توجه کنید که در این مسئله یک زبان ساده و جدید طراحی شد.

۲. این برنامه در واقع یک کامپایلر است که برنامه به زبان مورد نظر را به زبان C++ تبدیل می کند. با کامپایل خروجی ، برنامه اجرایی تولید می شود.

با کامپایل این برنامه و اجرای برنامه، عدد ۱۰ چاپ خواهد شد که نتیجه برنامه مبدا است.

۱۲. برنامه‌های مربوط به تولید مفسر و کامپایلر را طوری تغییر دهید که اگر در برنامه ورودی خطایی وجود داشت آن را گزارش کند. به عنوان مثال هر دو برنامه باید با دریافت عبارات ذیل پیغام خطای مناسب ارائه کنند.

```
A = 2 * 4
B =3
C= A % B
B=2
A= B * C
```

در این عبارات استفاده از % غیر مجاز است و همچنین انجام عملیات مستقیم روی اعداد نیز غیر مجاز است.

۱۳. نرم افزارهایی را نام ببرید که مانند کامپایلر و مفسرها، تحلیل لغوی انجام می‌دهند.

۱٤. نرم افزارهایی را نام ببرید که مانند کامپایلر و مفسرها، تحلیل نحوی انجام می‌دهند.

فصل دوم

تحلیلگر لغوی

هدف کلی

آشنایی با مفاهیم، تئوریها، تکنیکها، روشها و ابزارهای ساخت تحلیلگر لغوی

هدفهای رفتاری

پس از مطالعه این فصل انتظار می‌رود دانشجو بتواند:

۱- عملکرد تحلیلگر لغوی را بیان کند.

۲- نحوه ارتباط تحلیلگر لغوی با تحلیلگر نحوی را نشان دهد.

۳- جدول نمادها و نقش آن در کامپایلر را بیان کند.

٤- عبارات باقاعده را معرفی کند.

۵- با استفاده از عبارات باقاعده ساختار لغات زبان مبدا را بیان کند.

٦- برای عبارت با قاعده، NFA ایجاد کند.

۷- از NFA یک DFA ایجاد کند.

۸- برای عبارت با قاعده، مستقیما DFA تولید کند.

۹- برنامه ای جهت پذیرش رشته های تولیدی یک DFA ایجاد کند.

۱۰- با استفاده از زبانهای سطح بالا مانند C یک تحلیلگر لغوی ایجاد کند.

۱۱- با استفاده از ابزار تولیدکننده تحلیلگر لغوی یک تحلیلگر لغوی ایجاد کند.

۲-۱ مقدمه

در این فصل اولین فاز جلوبندی کامپایلر یعنی تحلیلگر لغوی را مورد بررسی قرار می‌دهیم و تکنیکها، روشها و ابزارهای ساخت تحلیلگر لغوی را معرفی می‌کنیم. تحلیلگر لغوی تنها فازی است که مستقیما با برنامه مبدا در ارتباط است. این فاز، برنامه ورودی را بررسی می‌کند، لغات برنامه مبدا و نوع آنها را تشخیص داده و برای تحلیلگر نحوی ارسال می‌کند.

در این فصل ابتدا انواع لغات یک برنامه را معرفی می‌کنیم و سپس با استفاده از عبارات با قاعده ساختار هر نوع لغت را توصیف می‌کنیم. در مرحله بعد عبارات با قاعده را به DFA تبدیل می‌کنیم و با استفاده از DFA ها تحلیلگر لغوی را ایجاد می‌کنیم. در نهایت ابزار flex را جهت تولید تحلیلگر لغوی معرفی می‌کنیم.

۲-۲ انواع لغات برنامه

آنچه برنامه نویس هنگام نوشتن برنامه خود می‌بیند با آنچه در اختیار تحلیلگر لغوی قرار می‌گیرد متفاوت است.

مثال ۲-۱ به عنوان مثال فرض کنیم برنامه نویسی، برنامه ذیل را به زبان پاسکال نوشته باشد.

```
program sample;
var i:integer;
    begin
        i:=2;
        writeln(i);
   end.
```

این برنامه به صورت جریانی از کاراکترها به صورت ذیل در اختیار تحلیلگر لغوی قرار می‌گیرد.

p	r	o	g	r	a	m		s	a	m	p	l	e	;	\n	v	a	r		i
:	i	n	t	e	g	e	r	;					b	e	g	i	n			
		i	:	=	2	;										w	r	i	t	e
l	n	(	i	)	;	\n					e	n	d.	\z						

شکل ۲-۱ نمایش برنامه در حافظه

همانطور که شکل ۲-۱ نشان می‌دهد ویرایشگر در مکانهایی که برنامه نویس کلید Enter زده است کاراکتر کنترلی \n و برای نشان دادن آخر فایل کاراکتر کنترلی \z درج کرده است. تحلیلگر لغوی برنامه مبدا را به صورت جریانی از کاراکترها(کنترلی و غیر کنترلی) می‌خواند و لغات را استخراج می‌کند. از آنجاییکه تحلیلگر لغوی برنامه ورودی را پویش

می‌کند، این فاز پویش‌گر[1] نیز نامیده می‌شود. به منظور شناخت و درک دقیق عملکرد تحلیلگر لغوی انواع لغات موجود در برنامه مبدا را معرفی می‌کنیم. لغات موجود در برنامه مبدا به چند دسته عمده ذیل تقسیم می‌شوند که عبارتند از:

- **کلمات کلیدی:** لغاتی که در زبان مبدا به مفهوم خاصی به کار می‌رود. مانند:

main,if,while,class,private,float در زبان c++

program,repeat,array,integer در زبان پاسکال

برخی از زبانهای برنامه‌نویسی مانند PL/1 اجازه استفاده از کلمات کلیدی به عنوان شناسه را می‌دهند. اما برخی دیگر از زبانها مانند پاسکال و C اجازه استفاده از کلمات کلیدی را به عنوان شناسه نمی‌دهند در این زبانها، کلمات کلیدی را کلمات رزرو شده[2] می‌نامند. به عنوان مثال کامپایلر C از قطعه برنامه ذیل خطا می‌گیرد. زیرا کلمه while کلمه کلیدی رزرو شده است.

```
int while,a;
```

- **علائم:** علامتهایی که به منظور نشان‌گذاری خاصی در زبان مبدا بکار می‌روند. مانند:

(,) , { , } و ; در زبان c++

(و) , ; . در زبان پاسکال

- **عملگرها:** علامتهایی که به منظور نشان دادن عملیات خاصی در زبان مبدا بکار می‌روند، مانند:

<,>,!=,+,?,*,% در زبان c++

<>,+ در زبان پاسکال

- **شناسه‌ها:** نامهایی که جزء کلمات کلیدی زبان مبدا نبوده و توسط برنامه‌نویس تعریف می‌گردند مانند: نام متغیرها، نام آرایه‌ها، نام توابع و نام کلاسها و برچسبها

- **ثوابت:** مقادیر عددی، کاراکتری، رشته‌ای و یا منطقی که برنامه نویس در برنامه از آنها استفاده می‌کند. مانند: عدد 342.67, 123، رشته"temp" وکاراکتر'a'.

- **توضیحات**[3]: قسمتی از برنامه مبدا است که توسط کامپایلر در نظر گرفته نمی‌شود. توضیحات به منظور افزایش خوانایی در برنامه قرار داده می‌شود و در عملکرد برنامه تاثیری ندارند.

---

1. Scanner 2. Reserved Word 3. Comments

- **فضاهای خالی:** تاثیر فضای خالی در برنامه بستگی به زبان مبدا دارد. در برخی زبانها مانند زبانC و پاسکال فضای خالی به منظور خواناتر کردن برنامه و جدا کردن لغات از یکدیگر به کار می‌رود. به عنوان مثال دو برنامه ذیل در زبان پاسکال معادل یکدیگر هستند.

جدول ۲-۱ فضای خالی در برنامه

برنامه ۱	برنامه ۲

program j;
var i:integer;
begin
i:=2;
writeln(i);
end.
	program j;var i:integer;begin i:=2; writeln(i);end.

مثال ۲-۲ برای درک بهتر عملکرد تحلیلگر لغوی برنامه ذیل را در نظر می‌گیریم.

```
program p1;{this is sample}
var  i:integer;
     j:real;
     str:string[20];
     4temp:integer;
begin
          j:=2.1;
          i:= 2*j;
          str:='hello world';
           k:=12;
ed.
```

تحلیلگر لغوی این برنامه را بررسی کرده و لغات ذیل را استخراج می‌کند.

جدول ۲-۲ لغات برنامه

لغت	نوع لغت
program	کلمه کلید program
p1	شناسه p1
;	علامت ;
{this is sample}	توضیحات
var	کلمه کلیدی var
i	شناسه i
:	علامت:
integer	کلمه کلیدی integer
j	شناسه j
:	علامت:
real	کلمه کلیدی real
;	علامت ;

str	شناسه str
:	علامت :
string	کلمه کلیدی string
[	علامت [
20	عدد ثابت ۲۰
]	علامت]
;	علامت ;
4temp	***خطا***
:	علامت :
integer	کلمه کلیدی integer
;	علامت ;
begin	کلمه کلیدی begin
j	شناسه j
:=	علامت انتساب :=
2.1	ثابت اعشاری 2.1
;	علامت ;
i	شناسه i
:=	علامت انتساب
2	ثابت عددی ۲
*	عملگر *
j	شناسه j
;	علامت ;
str	شناسه str
:=	علامت انتساب :=
'hello world'	ثابت رشته ای 'hello world'
;	علامت ;
k	شناسه k
:=	علامت انتساب:=
12	ثابت عددی 12
;	علامت ;
ed	شناسه ed
.	علامت .

تعیین یک لغت از جریان کاراکترهای ورودی پیچیدگیهای خاص خود را دارد. به عنوان مثال در جدول ۲-۲ دنباله 2.1 سه لغت عدد ثابت 2،عدد ثابت 1 و علامت نقطه در نظر گرفته

نمی‌شود بلکه یک لغت از نوع عدد ثابت اعشاری در نظر گرفته شده است و دنباله 'hello world' نیز به جای ٤ لغت، یک لغت و =: نیز یک لغت در نظر شده است.

در دنباله ذیل همه نمادها بدون فاصله در کنار یکدیگر قرار گرفته‌اند. تحلیلگر لغوی این دنباله را به چهار لغت شامل عدد اعشاری 2.1 ، علامت ; وشناسه j و علامت:= تجزیه می‌کند.

```
j:=2.1;
```

اگر دنباله فوق به صورت ذیل نوشته شود.

```
j     :=     2.1;
```

در عملکرد تحلیلگر لغوی تغییری بروز نمی‌کند و تحلیلگر لغوی باز هم این دنباله را به چهار لغت تجزیه می‌کند. در این مثال اضافه شدن فضای خالی بی‌تاثیر است. به عنوان مثال دیگر تحلیلگر لغوی دنباله ذیل را به یک لغت تجزیه می‌کند.

```
2.1
```

اما اگر این دنباله را به صورت ذیل تغییر دهیم.

```
2     1
```

تحلیلگر لغوی برای چنین دنباله‌ای سه لغت شامل عدد ثابت1 و علامت نقطه و عدد ثابت 2 را تشخیص می‌دهد[1]. تحلیلگر لغوی ساختار تک تک لغات را جدا از هم بررسی می‌کند. بررسی موارد دیگر از جمله ترتیب لغات بر عهده فازهای بعدی کامپایلر است. به عنوان مثال در برنامه مثال ۲-۲ ، تحلیلگر لغوی در دنباله k:=12; لغت k را به عنوان شناسه، شناسایی می‌کند. در حالیکه متغیر k قبلا معرفی نشده است. این خطا مربوط به ساختار لغت k نیست در نتیجه تحلیلگر لغوی خطای استفاده از متغیری که تعریف نشده است، را کشف نمی‌کند. همین مطلب در مورد لغت ed که در پایان برنامه درج شده است نیز صادق است، این لغت به تنهایی دارای ساختار صحیح لغوی مطابق شناسه می‌باشد، در نتیجه تحلیلگر لغوی دنباله ed را (به جای کلمه کلیدی end که به اشتباه ed تایپ شده است) شناسه در نظر می‌گیرد و اعلان خطا نمی‌کند. اگر لغتی با ساختارهای لغوی مورد انتظار تحلیلگر لغوی مطابق نباشد تحلیلگر لغوی آن را به عنوان خطا در نظر می‌گیرد. در برنامه مثال ۲-۲ دنباله 4temp از نظر تحلیلگر لغوی غیر مجاز است زیرا در زبان پاسکال هیچ لغتی نمی‌تواند با عدد شروع شود مگر آنکه تمام کاراکترهای دنباله آن نیز رقم باشد.

---

۱. البته باید دقت نمود که این مثال با توجه به مشخصات زبان پاسکال بررسی شده است.

مثال ۲-۳ در ذیل برنامه‌ای به زبان پاسکال ارائه شده است که از نظر تحلیلگر لغوی(نه فازهای دیگر) کاملا صحیح است!؟

```
end
program2 ;
var  A[1.1] + BB <>
end1
end
end j = 23 .
```

همانطور که ملاحظه شد از نظر تحلیلگر لغوی برنامه مثال ۲-۲ دارای خطا بود، اما برنامه مثال ۲-۳ از نظر تحلیلگر لغوی هیچ خطایی ندارد؟!

عملکرد تحلیلگر لغوی هنگام برخورد با یک خطای لغوی بستگی به نحوه پیاده سازی تحلیلگر لغوی دارد. تحلیلگر لغوی با خطا به چند صورت برخورد می‌کند که عبارتند از:

۱- **توقف:** در این روش تحلیلگر لغوی پس از کشف اولین خطا متوقف می‌شود. برنامه‌نویس خطا را رفع و برنامه را کامپایل می‌کند. تحلیلگر لغوی برنامه را از ابتدا بررسی می‌کند، اگر به خطای دیگری برخورد کرد دوباره متوقف می‌شود. عیب این روش افزایش تعداد دفعات انجام کامپایل است در این روش تحلیلگر لغوی برای هر خطا یکبار متوقف می‌شود بنابراین تعداد دفعاتی که نیاز به کامپایل مجدد است افزایش می‌یابد در نتیجه زمان تولید نرم افزار افزایش می‌یابد.

۲- **پوشش خطا:** تحلیلگر لغوی پس از کشف یک خطای لغوی متوقف نمی‌شود بلکه به کار خود ادامه می‌دهد. در این روش پس از کشف خطا، تحلیلگر لغوی از خطا چشم‌پوشی کرده و به بررسی بقیه برنامه ادامه می‌دهد و خطاهای دیگر را نیز کشف و گزارش می‌کند. یکی از روشهای پوشش خطا حذف کاراکترهای ورودی است تا زمانیکه، یک لغت معتبر کشف شود[۱].

اگر بتوان خطاهای لغوی را پیش بینی کرد، تحلیلگر لغوی می‌تواند لغت نامعتبر را کشف و پیغام خطای مناسب را گزارش داده و به کار خود ادامه دهد. به عنوان مثال یکی از انواع خطاهایی که معمولا رخ می‌دهد، شناسه‌ای است که با عدد شروع می‌شود. به همین جهت می‌توان تحلیلگر لغوی را به گونه‌ای ایجاد کرد که اینگونه دنباله‌ها را به عنوان لغت در نظر بگیرد ولی به جای ارسال آنها به تحلیلگر نحوی پیغام خطا به کاربر ارائه داده و به پویش ورودی بدون توقف ادامه دهد.

---

۱. این روش Panic Mode نامیده می شود.

۲-۳ نشانه

فاز بعد از تحلیلگر لغوی، تحلیلگر نحوی است که باید علاوه بر خود لغات، نوع هر لغت را نیز در اختیار داشته باشد. به همین جهت لازم است هماهنگی‌هایی بین تحلیلگر لغوی و تحلیلگر نحوی انجام گیرد. به همین منظور از نشانه[1] استفاده می‌گردد. نشانه، علامتی است که تحلیلگر لغوی و تحلیلگر نحوی برای مشخص کردن یک نوع خاص از لغات با یکدیگر توافق کرده‌اند. به عنوان مثال ممکن است تحلیلگر نحوی و تحلیلگر لغوی برای نشان دادن لغاتی از نوع شناسه از عدد ۲۵۷ استفاده کنند، به این ترتیب هرگاه تحلیلگر لغوی یک لغت از نوع شناسه را شناسایی می‌کند، عدد ۲۵۷ را برای تحلیلگر نحوی ارسال می‌کند. تحلیلگر نحوی نیز با دریافت عدد ۲۵۷ وجود یک لغت از نوع شناسه در برنامه ورودی را تشخیص می‌دهد. به عنوان مثال دیگر، ممکن است بین تحلیلگر نحوی و تحلیلگر لغوی برای نشان دادن ثوابت عددی، از ۲۵۸ استفاده گردد. هرگاه تحلیلگر لغوی، یک ثابت عددی را کشف کند عدد ۲۵۸ را به تحلیلگر نحوی ارسال می‌کند.

تحلیلگر لغوی پس از تشخیص هر لغتی نوع آن را تشخیص داده و نشانه توافق شده را برای تحلیلگر نحوی ارسال می‌کند. به عنوان مثال اگر تحلیلگر لغوی در برنامه مبدا لغات sum,count,average,t4 را کشف کند برای هر یک از این لغات عدد 257 (با فرض اینکه عدد 257 نشانه، شناسه باشد) را ارسال خواهد کرد، یعنی تحلیلگر لغوی ٤ بار عدد 257 را ارسال می‌کند. یک نوع لغت (مانند شناسه) ممکن است دارای چندین نمونه باشد. به عنوان مثال اگر در برنامه‌ای دنباله‌های 12,123,34275 موجود باشد تحلیلگر لغوی سه بار نشانه 258 را ارسال می‌کند. در برخی از موارد که یک نوع لغت فقط دارای یک نمونه است می‌توان از خود لغت به عنوان نشانه (یا علامت توافق شده) استفاده کرد. به عنوان مثال برای لغت = می‌توان از کد اسکی کاراکتر = به عنوان نشانه توافق شده استفاده کرد. در این موارد کد اسکی، به عنوان نشانه به کار می‌رود. از آنجاییکه کد اسکی کاراکترها حداکثر ۲۵۵ است به همین جهت نشانه استفاده شده برای دیگر لغات را بزرگتر از ۲۵۵ در نظر می‌گیرند.

طبق اصول برنامه نویسی هرگاه مقادیر ثابتی در برنامه استفاده شوند بهتر است از ثوابت استفاده گردد. به عنوان مثال عدد ۲۵۷ را با ID و ۲۵۸ را با NUM نشان می‌دهیم. اگر از زبان C برای پیاده‌سازی کامپایلر استفاده می‌شود ، می‌توان نشانه‌ها را به صورت ذیل تعریف کرد. در زبان C به صورت ذیل تعریف می‌شود.

```
#define  ID      257
#define  NUM     258
```

---

2. Token

```
#define  ASSIGN   128
```

در زبان پاسکال به صورت ذیل تعریف می‌شود.

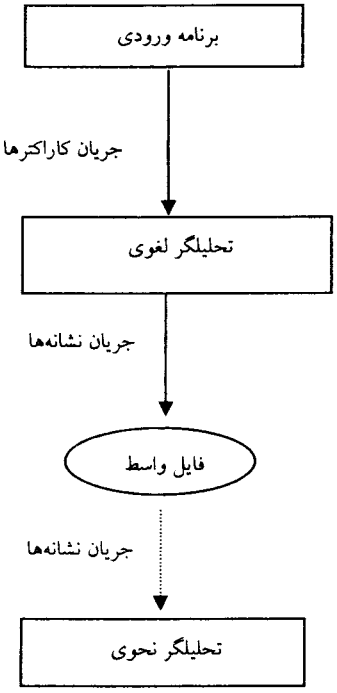
```
const          ID=257
const          NUM=258
const          ASSIGN =128
```

نام نشانه نیز در اختیار کامپایلر نویس است به عنوان مثال برای کلمه کلیدی if می‌توان نامهای مختلفی اختصاص داد. به عنوان مثال می‌توان از هر یک از نامهای ذیل استفاده کرد:

```
#define  ID            257
#define  ID_TK         257
#define  ID_TOKEN      257
```

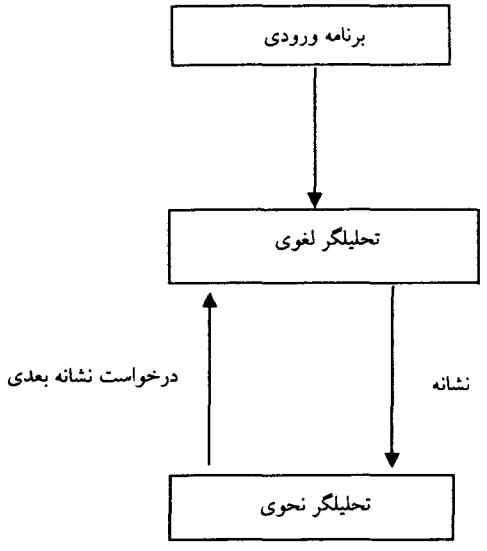
با توجه به مطالب ذکر شده تحلیلگر لغوی جریانی از کاراکترها را به عنوان ورودی دریافت کرده و جریانی از نشانه‌ها را به عنوان خروجی تولید کرده و برای تحلیلگر نحوی ارسال می‌کند. نحوه ارسال نشانه ها به تحلیلگر نحوی به دو روش قابل انجام است.

۱- استفاده از فایل واسط: در این روش تحلیلگر لغوی تمام فایل ورودی را خوانده و همه نشانه‌های لازم را تولید می‌کند و در یک فایل واسط قرار می‌دهد. سپس تحلیلگر نحوی این فایل را می‌خواند. شکل ۲-۲ این ساختار را نشان می‌دهد.



شکل ۲-۲ استفاده از فایل برای ارتباط تحلیلگر نحوی و لغوی

۲- ارتباط مستقیم: در این روش تحلیلگر نحوی از تحلیلگر لغوی درخواست نشانه می‌کند و تحلیلگر لغوی جریان ورودی را برای یافتن لغت بعدی پویش کرده و پس از تشخیص لغت بعدی، نشانه مربوطه را به تحلیلگر نحوی باز می‌گرداند. این روش سریعتر از روش قبل است به همین جهت اغلب کامپایلرها از این روش استفاده می‌کنند.



شکل ۲-۳ ارتباط مستقیم تحلیلگر لغوی و نحوی

۲-٤ نحوه تخصیص نشانه‌ها به انواع لغات

در این بخش نحوه تخصیص نشانه به انواع لغات برنامه را بررسی می‌کنیم.

۱- کلمات کلیدی: برای هر یک از کلمات کلیدی یک نشانه مجزا در نظر گرفته می‌شود. به عنوان مثال برای هر یک از کلمات کلیدی if, while, then, program یک نشانه مجزا تعریف می‌شود.

```
#define   IF_TK          259    //  برای کلمه کلیدی if
#define   WHILE_TK       260    //  برای کلمه کلیدی while
#define   PROGRAM_TK     261    //  برای کلمه کلیدی program
#define   THEN_TK        262    //  برای کلمه کلیدی then
```

۲- علائم: برای هر یک از علائم می‌توان یک نشانه مجزا در نظر گرفت. اگر علامت یک کاراکتری باشد می‌توان از کد اسکی علامت به عنوان نشانه استفاده کرد. مانند:

```
#define  ASSIGN_TK        61      //= برای علامت
#define  PARANTES1_TK     40      // ) برای علامت
#define  PARANTES2_TK     41      // ( برای علامت
#define  DOT_TK           46      // . برای علامت
```

۳- عملگرها: برای هر یک از عملگرها، می‌توان یک نشانه مجزا در نظر گرفت. اگر عملگر یک کاراکتری باشد می‌توان از کد اسکی عملگر به عنوان نشانه استفاده کرد. در ذیل چند نمونه از تخصیص نشانه به عملگرها نشان داده شده است:

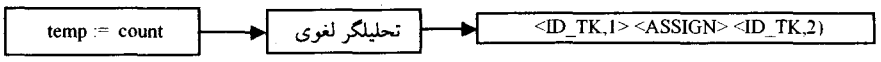
```
#define  GARTE_TK         62      //> برای علامت
#define  LESS_TK          60      //< برای علامت
#define  LESS_EQU__TK     265     //<= برای علامت
#define  GRATE_EQU_TK     266     //>= برای علامت
#define  EQU_TK           61      // =برای علامت
#define  NOT_EQU__TK      268     // <>برای علامت
```

همانطور که ملاحظه می‌شود برای هر عملگر و هر علامتی یک نشانه مجزا در نظر گرفته می‌شود. روش دیگری نیز وجود دارد و آن استفاده از یک نشانه برای چند عملگر از یک نوع است. به عنوان مثال برای عملگرهای مقایسه‌ای مانند <>,=,>,<=,< و < می‌توان از یک نشانه (مانند RELATION_TK) استفاده کرد. وقتی تحلیلگر لغوی یکی از عملگرها را تشخیص می‌دهد نشانهRELATION_TK را به تحلیلگر نحوی ارسال می‌نماید. ولی مسئله این است که تحلیلگر نحوی چگونه با دریافت نشانه یکسان تشخیص می‌دهد که در برنامه مبدا کدام یک از عملگرها وجود داشته است. برای حل این مشکل، تحلیلگر لغوی باید علاوه برای نشانه، خود عملگر را نیز به تحلیلگر نحوی ارسال کند یعنی به صورت<RELATION_TK,'='> <RELATION_TK,'<>'> و <RELATION_TK,'>='> و ... استفاده از روش دوم ساخت تحلیلگر نحوی را ساده‌تر خواهد کرد.

٤- شناسه‌ها: تحلیلگر لغوی برای شناسه‌ها فقط یک نشانه در نظر می‌گیرد. تحلیلگر لغوی وقتی یک شناسه را تشخیص می‌دهد، نشانه مربوط به شناسه‌ها (در مثال ما نشانهID_TK) را به تحلیلگر نحوی ارسال می‌کند. اگر در برنامه ای شناسه‌های temp,sum,average وجود داشته باشد تحلیلگر لغوی با شناسایی هر شناسه نشانه ID_TK را ارسال می‌کند. شناسه‌ها

علاوه بر تحلیلگر نحوی در فازهای بعدی نیز مورد نیاز است بنابراین از ساختاری به نام جدول نماد استفاده می‌شود. تحلیلگر لغوی شناسه‌ها را در جدول نماد ذخیره کرده و آدرس مدخل (به عنوان مثال اگر جدول نماد به صورت آرایه پیاده سازی شود آدرس مدخل، اندیس سطری از جدول نماد است که شناسه در آن ذخیره شده است) آن را به تحلیلگر نحوی ارسال می‌کند. این جدول برای فازهای بعدی نیز مورد استفاده است. اگر تحلیلگر لغوی عبارات temp:=count را در ورودی ببیند. نشانه‌هایی را به صورت نشان داده شده در شکل ۲-٤ به تحلیلگر نحوی ارسال می‌کند.

شکل ۲-٤ تبدیل برنامه به جریان نشانه ها با استفاده از جدول نماد



جدول ۲-۳ جدول نماد نمونه

اندیس	لغت
1	temp
2	Count

جدول ۲-۳ یک جدول نماد را به صورت ساده نشان می‌دهد.

۵- ثوابت: تحلیلگر لغوی هنگام تشخیص هر یک از انواع ثوابت یک نشانه مطابق نوع ثابت یافت شده به تحلیلگر نحوی (مانند NUM-TK برای اعداد صحیح) ارسال می‌کند. برای آنکه فازهای بعدی به مقدار ثوابت نیز دسترسی داشته باشند. تحلیلگر لغوی علاوه بر نشانه، مقدار ثابت را نیز همراه نشانه به تحلیلگر نحوی ارسال می‌کند. در تحلیلگر لغوی برای انواع ثوابت نشانه‌های مختلفی در نظر گرفته می‌شود به عنوان مثال در برخی تحلیلگرها برای اعداد اعشاری و صحیح دو نشانه مختلف در نظر می‌گیرند. یکی از عملیاتهای جنبی تحلیلگر لغوی تبدیل رشته عددی به یک عدد است. یعنی تحلیلگر لغوی با دریافت دنباله 123 ، این دنباله را به عدد 123 تبدیل می‌کند.

۶- فضاهای خالی و توضیحات: فضاهای خالی و توضیحات تاثیری در عملکرد برنامه ندارند، در نتیجه تحلیلگر لغوی با تشخیص فضاهای خالی و توضیحات، نشانه‌ای را برای تحلیلگر نحوی ارسال نمی‌کند و بدین ترتیب این گونه موارد حذف می‌شوند.

۲-۵ جدول نماد[1]

در بخش قبل بیان شد که تحلیلگر لغوی ممکن است با تشخیص مجموعه‌ای از لغات فقط یک نشانه را ارسال کند (برای برخی نشا نه ها، رابطه نشانه به لغات یک به چند است). لغات در فازهای بعدی مورد نیاز هستند، بنابراین تحلیلگر لغوی، لغات تشخیص داده شده را درون ساختاری به نام جدول نماد ذخیره می‌کند و آدرس آن را برای تحلیلگر نحوی ارسال می‌کند. جدول نماد برای هر شناسه یک رکورد در نظر می‌گیرد.

وقتی تحلیلگر لغوی یک شناسه را تشخیص می‌دهد. جدول نماد جستجو می‌شود تا مشخص شود آیا قبلا این شناسه در جدول نماد وارد شده است. اگر شناسه در جدول نماد نبود، شناسه جدید است و به جدول نماد اضافه می‌گردد. اطلاعات مربوط به نامها در طی انجام مراحل تحلیل لغوی و نحوی وارد جدول نماد می‌گردد. این اطلاعات در مراحل بعدی مانند تحلیل معنایی و تولید کد مورد استفاده قرار می‌گیرند. فازهای مختلف کامپایلر برای انجام فعالیتهای خود نیاز به اطلاعات کاملی در مورد نامها (مانند نام متغیرها، نام آرایه ها و نام روالها) دارند. اطلاعات مربوط به نامها در مراحل مختلف توسط فازهای مختلف کامپایلر جمع‌آوری می‌شود چنین اطلاعاتی در جدول نمادها ذخیره می‌شوند، تا در مراحل بعدی (مانند فاز تولید کد) مورد استفاده قرار گیرد. در ذیل برخی از انواع نامها و صفات آنها که در جدول نمادها ثبت می‌گردد، بررسی می‌شود.

متغیرها: نامهایی که برای نگهداری اطلاعات به کار می‌روند. مانند:

```
var temp:integer;
    sum:real;
```

در مورد temp ,sum نوع (یعنی real و integer)، دامنه اعتبار (محلی یا سراسری بودن) آنها در جدول نمادها ثبت می‌شود.

نام زیر برنامه‌ها: هر زیر برنامه دارای نام است که باید به همراه مشخصات آنها ثبت شود. به عنوان مثال در زبان پاسکال:

```
function f1(a:integer):integer;
```

در مورد f1 اطلاعاتی از قبیل نوع زیر برنامه (funcion یا procedure) پارامترهای ارسالی به تابع و نوع مقدار بازگشتی در جدول نماد ثبت می‌شود.

آرایه ها: نوع، اندازه و ابعاد آرایه در جدول نماد ثبت می‌شود.

برچسبها: آدرس برچسب در جدول نماد ثبت می‌شود.

مثال۲-٤ نشانه‌های ارسالی به تحلیلگر نحوی و جدول نماد برنامه زیر را مشخص کنید.

---

1. Symbol Table

```
program p1;
var  i:integer;
     j:real;
     str:string[20];
begin
          j:=2.1;
          i:= 2*j;
          str:='hello world';
end.
```

نشانه‌های ارسالی به تحلیلگر نحوی در جدول ۲-۵ نشان داده شده است. جدول نماد در جدول ۲-۴ نشان داده شده است. بخش صفات توسط فازهای دیگر تکمیل می گردد.

جدول ۲-۴ جدول نماد

	لغت	صفات
1	P1	
2	I	
3	J	
4	Str	

جدول نماد مولفه مهم و پر کاربردی در کامپایلر است بنابراین کارایی پیاده سازی آن بسیار مهم است زیرا تمام فازهای کامپایلر به نحوی با جدول نماد در ارتباط هستند. برای پیاده‌سازی جدول نماد روشهای مختلفی وجود دارد که از مهمترین آنها عبارتند از:

- لیست پیوندی
- جدول درهم ساز

در ادامه به شرح مختصر هر یک می‌پردازیم.

لیست پیوندی

ساده‌ترین روش پیاده سازی جدول نماد ایجاد یک لیست پیوندی از رکوردهای مربوط به نام‌ها است. زمان جستجو در این روش خطی است. برای افزایش کارایی می توان از ساختار درختی نیز استفاده کرد.

جدول درهم ساز

مهمترین بخش درهم سازی تابع درهم ساز است. این تابع رشته‌ای (نام) را به عنوان ورودی دریافت کرده و یک عدد صحیح بین 0 تا N-1 را بازمی گرداند. این عدد به عنوان اندیس در آرایه N عنصری جدول درهم‌ساز استفاده می‌گردد. گاهی دو رشته مختلف به یک عدد صحیح درهم‌سازی می شوند. که این حالت را برخورد می‌نامیم. روشهای مختلفی مانند استفاده از باکت برای رفع برخورد وجود دارد.

جدول ۲-۵ جریان نشانه ها

لغت کشف شده	نوع لغت	مقادیر ارسالی به فاز بعدی
program	کلمه کلید program	<PROGRAM_TK>
p1	شناسه p1	<ID_TK,1>
;	علامت ;	<SEMICOLON>
i	شناسه I	<ID_TK,2>
:	علامت:	< COLON >
Integer	کلمه کلیدی integer	<INTEGER_TK>
j	شناسه j	<ID_TK,3>
:	علامت:	< COLON >
real	کلمه کلیدی real	<REAL_TK>
;	علامت ;	<SEMISCOLON>
str	شناسه str	<ID_TK,4>
:	علامت :	<DOTDOT>
string	کلمه کلیدی string	<STRING_TK>
[	علامت [	<RBRACL>
20	عدد ثابت ۲۰	<INT_TK, 20>
]	علامت]	<RBRACR>
;	علامت ;	<SEMISCOLON>
Begin	کلمه کلیدی Begin	<BEGIN_TK>
j	شناسه j	<ID_TK,3>
:=	علامت انتساب :=	<ASSIGN>
2.1	ثابت اعشاری 2.1	<REAL_NUM,2.1>
i	شناسه I	<ID_TK,2>
:=	علامت انتساب :=	<ASSIGN>
2	ثابت عددی ۲	<INT_NUM,2>
*	عملگر *	<MUL_TK>
j	شناسه j	<ID_TK,2>
;	علامت ;	<SEMISCOLON>
str	شناسه str	<ID_TK,4>
:=	علامت انتساب :=	<ASSIGN>
'hello world'	ثابت رشته ای 'hello world'	<STR_TK,'hello world'>
;	علامت ;	<SEMISCOLON>
end	کلمه کلیدی end	<END>
.	علامت .	<POINT>

۲-۶ الگوها[1]

در بخش قبل انواع لغات موجود در برنامه مبدا معرفی شدند. حال نیاز به برنامه‌ای است که جریان کاراکترهای ورودی را بررسی کرده و مشخص کند که در جریان کاراکترهای ورودی کدامیک از انواع لغات وجود دارد تا نشانه مربوطه را ارسال کند. برای انجام این بخش، ابتدا باید ساختار انواع لغات موجود در برنامه مبدا معرفی گردند. از الگو برای توصیف ساختار لغات استفاده می‌شود. الگو عبارتی است که ساختار مشترک مجموعه‌ای از لغات را بیان می‌کند. الگوها در انواع سیستمها نیز استفاده می‌شوند. به عنوان مثال در سیستم عامل DOS می‌توان از دستور ذیل استفاده کرد.

```
dir   *.doc
```

عبارت *.doc یک الگو است که ساختار نام فایلهایی (لغاتی) را معرفی می‌کند که دارای پسوند doc باشند. به عنوان مثال دیگر در برنامه find (یا search) درwindows برای یافتن فایلهایی که نامهای آنها سه کاراکتر و پسوند آنها BMP باشد از عبارت ???.BMP در قسمت name استفاده می‌شود. عبارت ???.BMP نیز یک الگو است.

عبارات باقاعده[2] ابزاری هستند که می‌توان برای توصیف الگو از آنها استفاده کرده و توسط آنها ساختار انواع لغات را معرفی کرد. به عنوان مثال به جای اینکه بگوییم عدد صحیح در زبان پاسکال چگونه ترکیبی است می‌توان از عبارت با قاعده استفاده نمود.

عبارات باقاعده در بحث نظریه زبانها و ماشینها به طور کامل مورد بررسی قرار می‌گیرد اما به جهت اهمیت آن در ساخت تحلیلگر لغوی در این قسمت مروری بر زبانها و عبارات باقاعده خواهیم داشت.

۲-۷ زبانها

ابتدا خصوصیات کلی زبانها و رشته‌ها را بیان کرده سپس به طور اخص به عبارات باقاعده می‌پردازیم. در ابتدا چند تعریف اولیه ارائه می‌دهیم.

الفبای زبان: به مجموعه‌ای ناتهی و متناهی از نمادهای ساده و غیر قابل تجزیه الفبای زبان می‌گویند. مجموعه الفبا را با Σ نشان می‌دهیم.

مانند: الفبای دودویی $\Sigma = \{0,1\}$

الفبای زبان انگلیسی $\Sigma = \{a,b,c.....z,A,B,C,...Z\}$

---

1. Pattern
2. Regular Expression

الفبای زبان برنامه نویس پاسکال Σ={a,b,c,d....A,B,c,..,[,],{,},^,*,.,...}

رشته: یک دنباله متناهی از نمادهای الفبا را رشته گویند. رشته تهی نیز رشته است زیرا دنباله‌ای با صفر نماد است. رشته تهی را با∈نشان می‌دهیم. مانند:

رشته های ∈، ۰۱۰۰، ۰۰۰، ۱۱۱، ۰۱، .... از الفبای دودویی Σ ={0,1}.

رشته‌های school، book، soosooosooos ..... از الفبای انگلیسی Σ={a,b,c.....z,A,B,C,...Z}.

رشته‌های a,ab,aba,aa,aba,bbbabbabb,a از الفبای Σ={a,b}.

تعداد نمادهای بکار رفته در رشته را طول رشته می‌نامیم. طول رشته‌ای مانندs را با |s| نشان می‌دهیم. به مثالهای ذیل توجه کنید.

|110010|=6
|aab|=3
|∈|=0

زبان: مجموعه‌ای از رشته‌ها را زبان گویند. به عنوان مثال هر یک از مجموعه‌های زیر یک زبان هستند.

۱- {11,1111,010001}

۲- {a,aab,aba,....}

۳- مجموعه رشته‌هایی که حداقل یک a دارند.

٤- مجموعه رشته‌هایی که حداکثر ۳ تا a دارند.

بر روی زبانها عملیات مختلفی تعریف می‌شود که در ذیل به تشریح برخی از آنها می‌پردازیم.

عملیات اجتماع

اجتماع دو زبان L و M را به صورت LUM نشان می‌دهیم که عبارت است از مجموعه ای از رشته‌ها که یا در زبان M یا در زبان L باشند. این تعریف را می‌توان به صورت زیر نیز نشان داد.

LUM={s|s∈M or s∈L}

مثال۲-۵ فرض کنید L={ab,aab,aaab,aaaab} زبان تعریف شده برروی الفبا Σ={a,b} باشد و M={12,21,32,23,43,34} زبان تعریف شده بر روی الفبای Σ= {1,2,3,4} باشد آنگاه LUM عبارت است از :

LUM={ ab,aab,aaab,aaaab, 12,21,32,23,43,34}.
MUL={ 12,21,32,23,43,34,ab,aab,aaab,aaaab}

با توجه به تعریف عملگر اجتماع و خواص مجموعه‌ها می‌توان نتیجه گرفت که عملگر اجتماع خاصیت جابجایی دارد. یعنی: LUM=MUL

هیچیک از رشته های ذیل در مجموعه LUM قرار ندارند، زیرا هیچیک به تنهایی نه درL قرار دارد نه در M.

12ab,aab21,2134,3421

عملیات الحاق

الحاق دو زبان L و M را به صورت L.M نشان می‌دهیم(برای اختصار می‌توان به صورت LM نیز نشان داد). زبان LM عبارت است از مجموعه ای از رشته‌ها که از ترکیب رشته ای در زبان L و رشته‌ای در زبان M به دست آمده باشند. این تعریف را می‌توان به صورت زیر نشان داد.

$LM=\{st|s\in L \text{ and } t\in M\}$

مثال۲-٦ فرض کنید زبان L={aa,aaaa,aaaaaa} زبان تعریف شده برروی الفبای$\Sigma=\{a\}$ باشد و M={12,21} زبان تعریف شده بر روی الفبای $\Sigma=\{1,2\}$ باشد آنگاه LM وML عبارت است از:

LM={ aa12,aa21,aaaa12,aaaa21,aaaaaa12,aaaaaa21}
ML={12aa,21aa,12aaaa,21aaaa,12aaaaaa,21aaaaaa}

با توجه به تعریف عملگر الحاق می‌توان نتیجه گرفت که عملگر الحاق خاصیت جابجایی ندارد یعنی:

$ML \neq LM$

دقت کنید که رشته aa متعلق به LM و ML نیست. زیرا از ترکیب رشته‌ای از M و رشته‌ای از L به دست نیامده است. همچنین رشته aa12 متعلق به زبان ML نیست. زیرا قسمت اول رشته aa12 یعنی aa متعلق به M نیست.

عملیات بستار

اگر L یک زبان باشد بستار L را به صورت L^* نشان می‌دهیم. L^* یعنی الحاق صفر یا بیشتر L با خودش. L^* را می‌توان به صورت زیر نشان داد.

$L^*=L.L.L....$

مثال۲-۷ اگر زبان L={ab,10} را در نظر بگیریمL^* به صورت ذیل خواهد بود.

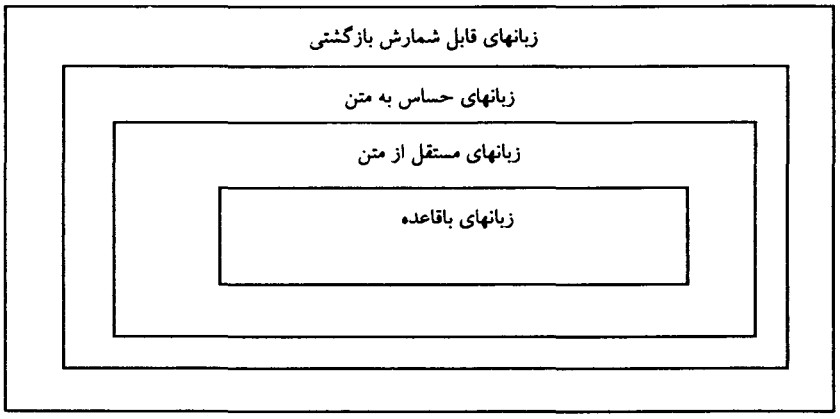
$L^0=\{\}$
$L^1=\{ab,10\}$
$L^2=l^1.l^1=\{abab,ab10,10ab,1010\}$
$L^3=L^2.L^1=\{ababab,ab10ab,10abab,1010ab, abab10,ab1010,10ab10,101010\}$
.
.
.

۲-۸ انواع زبانها

با استفاده از الفبای Σ تعداد زیادی زبان می‌توان تعریف کرد. برخی از این زبانها ویژگیهای خاصی دارند، که بر اساس این ویژگیها زبانها دسته‌بندی می‌شوند. انواع زبانها عبارتند ا ز:

۱- زبانهای باقاعده[1]
۲- زبانهای مستقل از متن[2]
۳- زبانهای وابسته به متن[3]
٤- زبانهای قابل شمارش بازگشتی[4]

شکل ۲-۵ رابطه بین این زبانها را نشان می‌دهد. به عنوان مثال زبانهای با قاعده زیر مجموعه‌ای از زبانهای مستقل از متن هستند به عبارت دیگر هر زبان باقاعده ای یک زبان مستقل از متن است اما عکس این مطلب صحیح نیست. از بین زبانهای ذکر شده در این بخش فقط به زبانهای باقاعده می‌پردازیم.



شکل ۲-۵ انواع زبانها

۲-۹ زبانهای باقاعده

زبان باقاعده زبانی است که بتوان آن را توسط یک عبارت باقاعده توصیف کرد. به جای بیان یک زبان با مجموعه‌ای از رشته‌ها، عبارت باقاعده، از الگویی برای بیان ویژگی زبان استفاده می‌کند. عبارات باقاعده به صورت ذیل تعریف می‌شوند:

---

1. Rgular Languages
2. Context free Language
3. Context Sensitive Languages
4. Recursively Enumerable Languages

۱- ∈ یک عبارت باقاعده است.

۲- به ازای هر a عضو Σ باشد، r=a یک عبارت باقاعده است که زبان {a} را مشخص می‌کند.

۳- اگر r=a یک عبارت باقاعده باشد آنگاه r* یک عبارت باقاعده است که زبان(L(r))* را مشخص می‌کند.

٤- اگر r1 و r2 عبارت باقاعده باشند و L(r1) و L(r2) به ترتیب زبانهای تولید شده توسط r1, r2باشند.آنگاه r=r1|r2 نیز یک عبارت باقاعده است که زبان L(r1)UL(r2) را تولید می‌کند.

مثال۲-۸ اگر r1=a* یک عبارت باقاعده باشد که زبان L(r1)={∈,a,aa,aaa,aaaa,...} و r2=b* نیز یک عبارت باقاعده که زبان L(r2)={ ∈,b,bb,bbb,...} آنگاه r=a*|b* نیز یک عبارت باقاعده است که زبان L(r1)Ul(r2)={ ∈,a,aa,aaa,...,b,bb,bbb,...} را تولید می‌کند.

۵- اگر r1 و r2 عبارت باقاعده باشند و L(r1) و L(r2) به ترتیب زبانهای تولید شده توسط r1, r2 باشند.آنگاه r=r1.r2 نیز یک عبارت باقاعده است که زبان L(r1).L(r2) را تولید می‌کند.

مثال۲-۹ اگر r1=a* یک عبارت باقاعده باشد که زبان L(r1)={∈,a,aa,aaa,aaaa,...} و r2=b* نیز یک عبارت باقاعده باشد که زبان L(r2)={ ∈,b,bb,bbb,...} را تولید کند، آنگاه r=a*.b* نیز یک عبارت باقاعده است که زبان ذیل را تولید می‌کند.

L(r1).L(r2)={ ∈,ab,abb,abbb,...,aa,aab,aabb,aabbb,...,aaab,aaabb,aaabbb,...}

در برخی موارد ارائه یک عبارت باقاعده ممکن است طولانی و پیچیده شود و در نتیجه درک و فهم آن مشکل شود. برای حل این مشکل می‌توان به هر عبارت باقاعده یک نام اختصاص داد و از آن در بیان عبارات باقاعده بعدی استفاده کرد. برای اختصاص نام برای یک عبارت باقاعده از روش ذیل استفاده می‌شود.

n1→r1
n2→r2
.
.

در این تعریف n_i نام مورد نظر و r_i عبارت باقاعده است. بعد از تعریف هر نام جدید می‌توان از آن برای ساخت عبارات باقاعده بعدی استفاده کرد. در واقع بعد از تعریف هر نام، آن نام مانند الفبای زبان قابل استفاده است.

مثال۲-۱۰

lower→a|b|c|d|e|f|g|h|...|z

upper→A|B|C|D|E|F|G|H|...|Z

بعد از تعریف lower و upper این دو نام مانند الفبای زبان برای تعریف عبارات باقاعده قابل استفاده هستند. هرجایی که این اسامی استفاده شوند عبارات باقاعده معادل آنها جایگزین می‌شود. مانند:

lower|upper این عبارت باقاعده معادل عبارت باقاعده ذیل است.

(a|b|c|d|e|f|g|h|...|z) | (A|B|C|D|E|F|G|H|...|Z)

همانطور که ملاحظه می‌شود عبارت باقاعده lower|upper از عبارت فوق ساده‌تر و کوتاهتر و قابل فهم‌تر است. در ذیل نمونه دیگری را ملاحظه می‌کنیم.

letter→lower|upper
digit→0|1|2|3|4|5|6|7|8|9
id→letter(letter|digit)*

۲-۱۰ ماشین خودکار متناهی

چگونه می‌توان تشخیص داد که یک رشته، متعلق به یک زبان است یا خیر. برای پاسخ به این سوال از تشخیص دهنده[1] استفاده می‌شود. تشخیص‌دهنده، یک ماشین متناهی است که یک رشته مانند x را دریافت کرده و در صورتی که رشته x متعلق به زبان باشد جواب بلی و در غیر این صورت جواب خیر می‌دهد. برای هر نوع زبان تشخیص دهنده خاصی وجود دارد. تشخیص دهنده زبانهای باقاعده یک ماشین خودکار متناهی است. ماشینهای خودکار متناهی دو نوع هستند که عبارتند از:

- ماشین خودکار قطعی(DFA)
- ماشین خودکار غیر قطعی(NFA)

در ذیل به تشریح هر یک می‌پردازیم.

ماشین خودکار قطعی را به صورت M=(Q,Σ,&,q0,F) نشان می‌دهیم که شامل بخشهای ذیل است:

۱- Q: مجموعه ای متناهی از حالات است.

۲- Σ : مجموعه‌ای متناهی از نمادها که الفبای زبان است.

۳- q0: یک حالت از حالات(q0∈Q) به عنوان حالت شروع در نظر گرفته می‌شود.

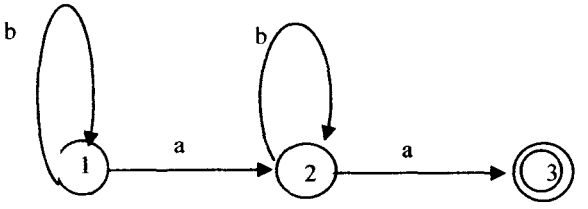
۴- F :زیر مجموعه ای از حالات (F⊂Q) به عنوان حالت نهایی در نظر گرفته می‌شود.

۵- &: یک تابع گذر است که نشان می‌دهد از هر حالت q(q∈Q) و به ازای هر یک از نمادهای الفبا (a∈Σ) به چه حالتی می‌رسد. در این تابع به ازای هر حالت و هر یک از الفبای زبان فقط یک حالت بعدی وجود دارد.

---

1. Recognizer

مثال۲-۱۱ شکل ذیل یک DFA را نشان می‌دهد.



شکل ۲-۶ DFA

در این DFA داریم:

Q={1,2,3}
Σ={a,b}
q0={1}
F={3}
&(1,a)=2
&(2,a)=3
&(1,b)=1
&(2,b)=2

همانطور که ملاحظه می‌شود تابع & به ازای هر حالت و یک نماد فقط یک حالت بعدی می‌دهد. اگر برچسب یالهای مسیری که از حالت شروع به حالت پایان می‌رسد را کنار یکدیگر قرار دهیم رشته‌ای تولید می‌شود که توسط DFA قابل تولید است. مجموعه رشته‌های قابل تولید DFA زبانی را تشکیل می دهد. این زبان توسط DFA پذیرفته می‌شود.

برای نمایش DFA در برنامه از جدول تغییر حالت استفاده می‌شود. جدول تغییر حالت جدولی است که در آن سطرها نشان دهنده حالات و ستونها نشان دهنده نمادها است. به عنوان مثال DFAی شکل ۲-۶ به صورت ذیل نشان داده می‌شود.

جدول ۲-۶ جدول تغییر حالت DFA

	a	b
1	2	1
2	3	2
3		

ماشین خودکار غیر قطعی را به صورت M=(Q,Σ,&,q0,F) نشان می‌دهیم که شامل بخشهای ذیل است:

۱- Q :مجموعه ای متناهی از حالات است.

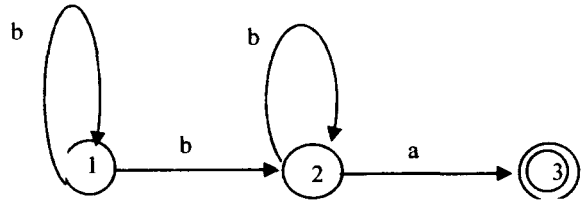
۲- Σ : مجموعه‌ای متناهی از الفبا است.

۳- q0: یک حالت ازمجموعه حالات($q0 \in Q$) به عنوان حالت شروع در نظر گرفته می‌شود.

۴- F :زیر مجموعه ای از حالات ($F \subset Q$) به عنوان حالات نهایی در نظر گرفته می‌شود.

۵- &: یک تابع گذر است که نشان می‌دهد از هر حالت q($q \in Q$) و به ازای هر یک از الفبا $a \in \Sigma$ به چه حالت یا حالتهایی می‌رود. در این تابع به ازای هر حالت و هر یک از الفبا ممکن است چند حالت بعدی وجود داشته باشد.

مثال۲-۱۲ در این NFA داریم:



شکل ۲-۷ NFA

Q={1,2,3}
Σ={a,b}
q0={1}
F={3}
&(1,b)={1,2}
&(2,a)=3
&(2,b)=2

همانطور که ملاحظه می‌شود تابع & به ازای حالت یک و نماد b بیش از یک حالت بعدی وجود دارد. در NFA نیز اگر برچسب یالهای مسیری که از حالت شروع به حالت پایان می‌رسد را در کنار یکدیگر قرار دهیم رشته‌ای تولید می‌شود. NFA را نیز می‌توان با استفاده از جدول نشان داد با این تفاوت که ممکن است در یک خانه چند حالت قرار بگیرد. NFAی شکل ۲-۷ را می‌توان به صورت ذیل نشان داد.

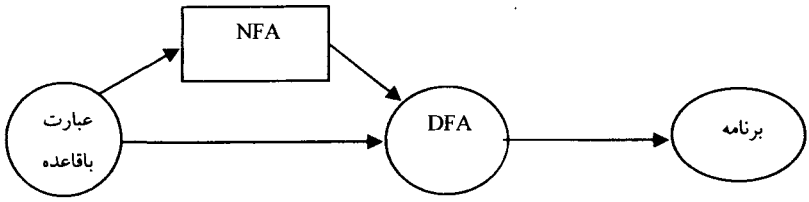
جدول ۲-۷ جدول تغییر حالت NFA

حالات DFA	a	b
1		1,2
2	3	2
3		

رشته x توسط DFA ویا NFA پذیرفته می شود اگر مسیری از حالت شروع به حالت پایان موجود باشد به طوریکه برچسب لبه‌ها x را تشکیل دهند. برای تولید برنامه‌ای که یک

لغت را دریافت کند و بررسی کند آیا توسط عبارات باقاعده r قابل تولید است یا خیر. روشهای متعددی وجود دارد. یکی از این روشها تبدیل عبارات باقاعده به DFA و سپس تبدیل DFA به برنامه است. به این منظور، ابتدا نحوه تبدیل عبارات با قاعده به DFA را نشان می‌دهیم و در بخش بعدی نحوه تبدیل DFA به برنامه را توضیح می‌دهیم.

برای تبدیل عبارات باقاعده به DFA نیز روشهای گوناگونی وجود دارد. یکی از این روشها تبدیل عبارات باقاعده به NFA و سپس تبدیل NFA به DFA است. روش دیگر تبدیل مستقیم عبارات باقاعده به DFA است. در بخش بعد، روش تبدیل عبارت باقاعده به NFA را مورد بررسی قرار می‌دهیم. این روندها در شکل ۲-۸ نشان داده شده است.



شکل ۲-۸ نحوه تبدیل عبارت باقاعده به برنامه

۲-۱۱- ایجاد NFA از عبارت باقاعده به روش تامپسون[1]

برای تبدیل عبارت باقاعده به NFA قوانین ذیل را روی عبارت با قاعده اعمال می‌شود.

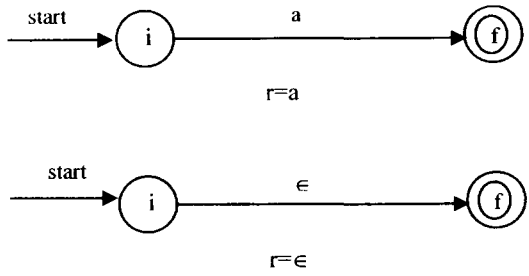
۱- عبارت با قاعده r را در نظر می‌گیریم. r را به نمادهای اولیه تجزیه می‌کنیم. در جدول ذیل چندین عبارت باقاعده و نحوه تجزیه آن را ملاحظه می‌کنیم.

جدول ۲-۸ تجزیه عبارت باقاعده به نمادهای اولیه

عبارت با قاعده	نمادهای اولیه عبارت با قاعده			
r= a|b	r1= a	r2=b		
r=(a|b)*(ca)*	r1=a	r2=b	r3=c	r4=a
r=(∈|b|c)*	r1=∈	r2=b	r3=c	

پس از شکستن عبارت باقاعده به عبارات باقاعده‌ای که فقط یک نماد دارند، برای هر یک از نمادها یک NFA می‌سازیم. برای ساخت این NFA یک حالت شروع به نام i و یک حالت نهایی به نام f ایجاد می‌کنیم. بین این دو حالت یک لبه با برچسب نماد مورد نظر ایجاد می‌کنیم. در مثال ذیل دو NFA برای r1=a و r2= ∈ نشان داده شده است.

1. Thompson

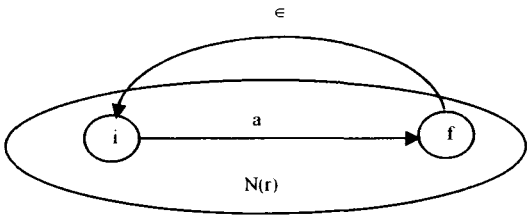


شکل ۲-۹ ایجاد NFA از نماد اولیه

پس از شکستن عبارات با قاعده به نمادهای اولیه و به دست آوردن NFA برای هر یک از آنها، NFA های به دست آمده را به وسیله قوانین ذیل به طور بازگشتی با یکدیگر ترکیب می‌کنیم تا NFAی نهایی به دست آید.

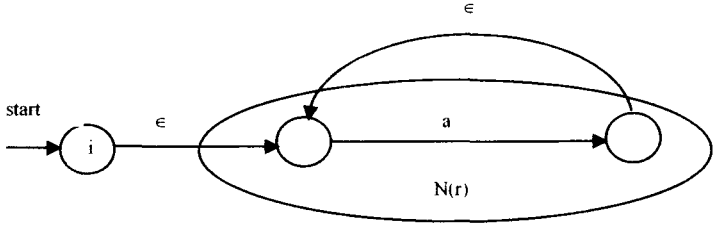
۲-۱- اگر N(r)، NFAی حاصل از عبارت باقاعده r باشد، آنگاه NFAی حاصل از r* را N(r*) می‌نامیم که به صورت ذیل ساخته می‌شود.

- از حالت پایان به حالت شروع N(r) یک لبه با برچسب ∈ رسم می‌کنیم. این لبه امکان تکرار را فراهم می‌کند.



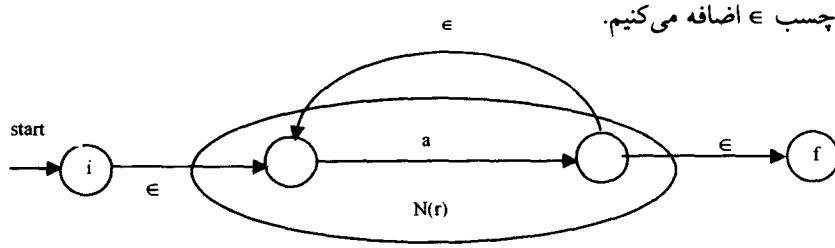
شکل ۲-۱۰ مرحله اول ساخت NFA برای a*

یک حالت شروع جدید به نام i ایجاد می‌کنیم. از حالت i به حالت شروع یک لبه با برچسب ∈ اضافه می‌کنیم.



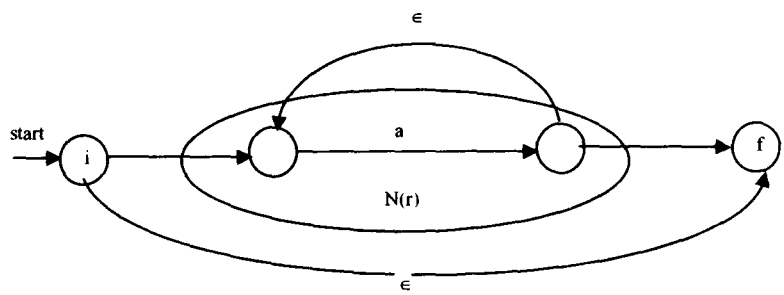
شکل ۲-۱۱ مرحله دوم ساخت NFA برای a*

همچنین یک حالت پایانی به نام f ایجاد می‌کنیم. از حالت پایانی N(r) به حالت f یک لبه با برچسب ∈ اضافه می‌کنیم.



شکل ۲-۱۲ مرحله سوم ساخت NFA برای a*

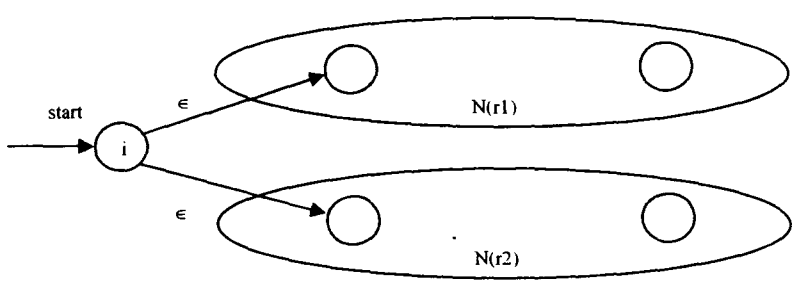
- از حالت i به حالت f یک انتقال با برچسب ∈ اضافه می‌کنیم. این لبه باعث می‌شود NFA بتواند ∈ را نیز تولید کند.



شکل ۲-۱۳ مرحله چهارم ساخت NFA برای a*

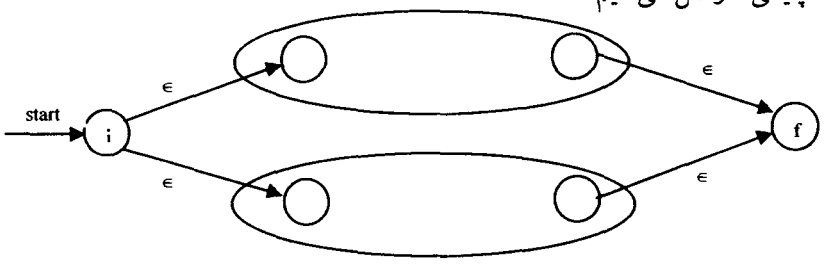
۲-۲- اگر N(r1) و N(r2)، NFAهای حاصل از عبارات باقاعده r1 و r2 باشند آنگاه NFAی حاصل از r1|r2 را N(r1|r2) می‌نامیم که به صورت ذیل ساخته می‌شود.

- یک حالت شروع به نام i ایجاد کرده و آن را با لبه‌ای با برچسب ∈ به حالات شروع N(r1) و N(r2) وصل می‌کنیم.



شکل ۲-۱۴ مرحله اول ساخت NFA برای r1|r2

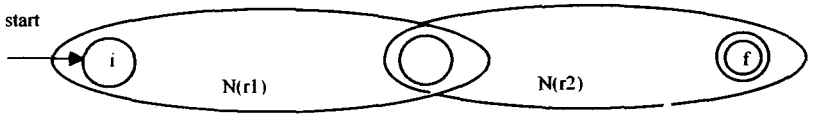
- یک حالت پایانی به نام f ایجاد کرده و از حالات پایانی N(r1) و N(r2) با انتقال ∈ به حالت پایانی f وصل می‌کنیم.



شکل ۲-۱۵ مرحله دوم ساخت NFA برای r1|r2

۲-۳- اگر N(r1) و N(r2)، NFAهای حاصل از عبارات باقاعده r1 و r2 باشند آنگاه NFAی حاصل از r2 . r1 را N(r1.r2) می‌نامیم که به صورت ذیل ساخته می‌شود.

- حالت شروع N(r1) را حالت شروع N(r1.r2) در نظر می‌گیریم.
- حالت پایان N(r1) را با حالت شروع N(r2) ادغام می‌کنیم.
- حالت پایان N(r2) را حالت پایانی N(r1.r2) در نظر می‌گیریم.



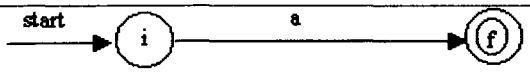
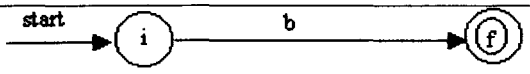
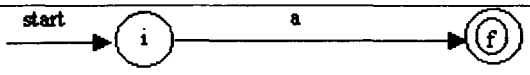
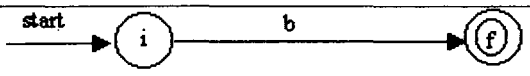
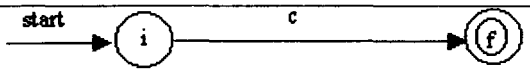
شکل ۲-۱۶ ساخت NFA برای r1.r2

در این قسمت با استفاده از روشی که ذکر شد عبارت باقاعده (ab)*(a|b)c را به یک NFA تبدیل می‌کنیم. ابتدا این عبارت را به نمادهای اولیه تجزیه می‌کنیم.

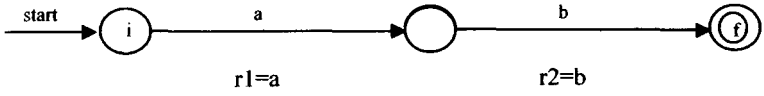
r1=a
r2=b
r3=a
r4=b
r5=c

همانطورکه ملاحظه می‌کنید ممکن است چند عبارت مانند هم تولید شود مانند: r1 و r3 یا r2 و r4. بعد از مرحله تجزیه هر عبارت را به یک NFA تبدیل می‌کنیم.

جدول ۲-۹ ساخت NFA برای نمادهای عبارت باقاعده

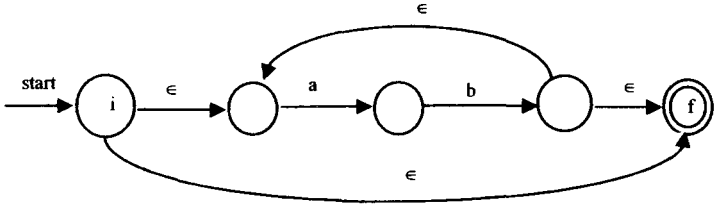
r1=a	
r2=b	
r3=a	
r4=b	
r5=c	

پس از تجزیه عبارت باقاعده به نمادهای اولیه و تبدیل آنها به NFA، مرحله ترکیب کردن آنها را شروع می‌کنیم. ابتدا NFAی مربوط به عبارت باقاعده s1=r1.r2=(a.b)=ab را می‌سازیم. در این مرحله N(r1) و N(r2) را به روشی که در الگوریتم ذکر شده بود ترکیب می‌کنیم.



شکل ۲-۱۷ NFA عبارت باقاعده s1=ab

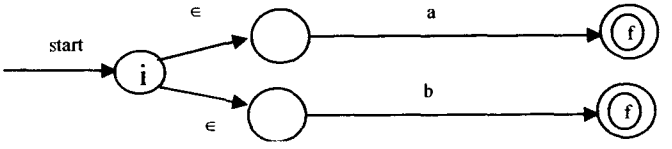
برای ساخت NFAی *(ab) تغییر ذیل را روی NFA(s1) ایجاد می‌کنیم.



شکل ۲-۱۸ NFAی مربوط به *(ab)

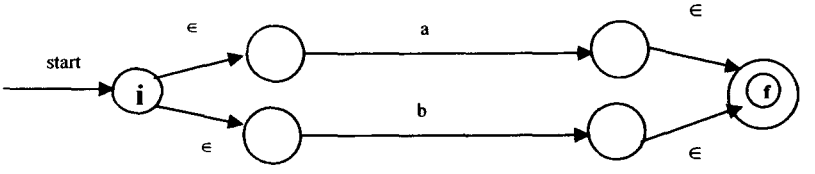
پس از ترکیب r1 و r2، در مرحله بعد، r3=a و r4=b را ترکیب می‌کنیم. با استفاده از الگوریتم ترکیب، NFA مربوطه به s2=r3|r4=a|b به صورت ذیل به دست می‌آید.

مرحله اول:



شکل ۲-۱۹: مرحله اول ساخت NFA مربوط به عبارت باقاعده a|b

مرحله دوم:

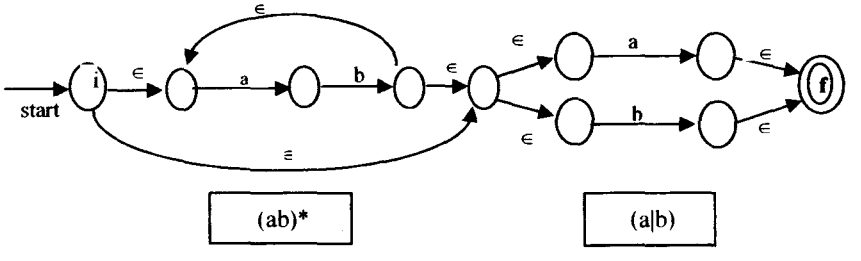


s3=a|b

شکل ۲-۲۰ مرحله دوم ساخت NFA مربوط به عبارت باقاعده a|b

پس از ساخت NFA(s1) و NFA(s2) این دو NFA را با یکدیگر ترکیب می‌کنیم. با توجه به رابطه ذیل، NFA(s3) را با ترکیب NFA(s1) و NFA(s2) محاسبه می‌کنیم.

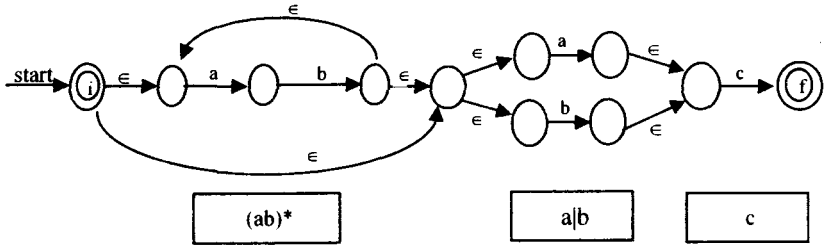
s3=s1.s2=(ab)*.(a|b)



شکل ۲-۲۱ NFA مربوط به عبارت باقاعده (ab)*(a|b)

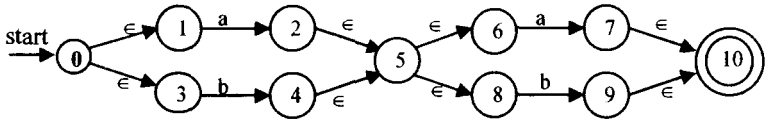
بعد از این مرحله NFA مربوط به عبارت باقاعده (ab)*(a|b) تکمیل گردیده است.

آخرین مرحله ترکیب این NFA با NFA مربوط به r5=c می‌باشد که این مرحله نیز در شکل ۲-۲۲ نشان داده شده است.



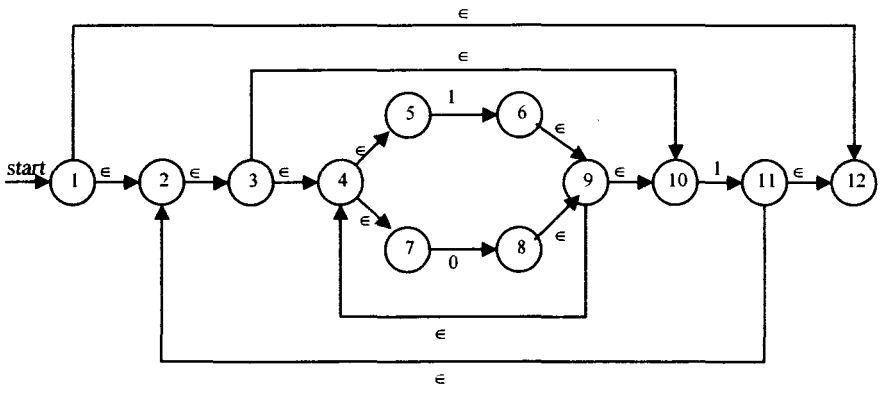
شکل ۲-۲۲ NFA مربوط به (ab)*(a|b)c

مثال۲-۱۳ برای عبارت باقاعده (a|b)(a|b) یک NFA بسازید.



شکل ۲-۲۳ NFA مربوط به عبارت باقاعده (a|b)(a|b)

مثال۲-۱٤ برای عبارت با قاعده *(1*(0|1)) یک NFA بسازید.



شکل ۲-۲٤ NFA مربوط به عبارت باقاعده *(1*(0|1))

با توجه به مراحل ساخت NFA به روش تامپسون، به ازای هر نماد عبارت باقاعده r دو حالت ایجاد می‌شود. همچنین به ازای هر عملگر | و* دو حالت ایجاد می‌گردد. عملگر الحاق حالت جدیدی به NFA اضافه نمی‌کند. در نتیجه تعداد حالات NFA تولید شده

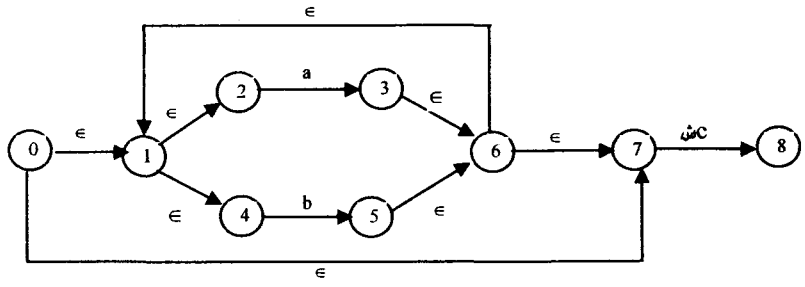
حداکثر دو برابر تعداد نمادها و عملگرهای عبارات باقاعده r یعنی 2|r| است. در نتیجه پیچیدگی فضای لازم برای ذخیره NFA ، O(|r|) است.

۲-۱۲ ایجاد DFA از NFA

برای تشریح ساخت DFA از روی NFA ابتدا چند تعریف ارائه می‌شود سپس با استفاده از این تعاریف الگوریتم ساخت DFA را از روی NFA بیان می‌گردد.

∈_closure(s): مجموعه حالاتی از NFA که از حالت s و با تبدیل ∈ قابل دسترسی است.

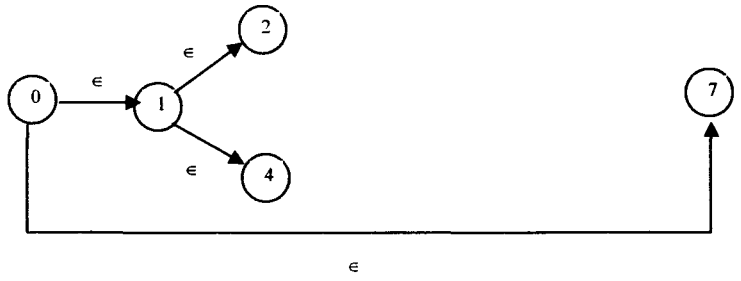
مثال۲-۱۵ با توجه به NFA زیر ∈_closure(0)، ∈_closure(3) و ∈_closure(7) را محاسبه کنید.



شکل ۲-۲۵ عبارت باقاعده (a|b)*c

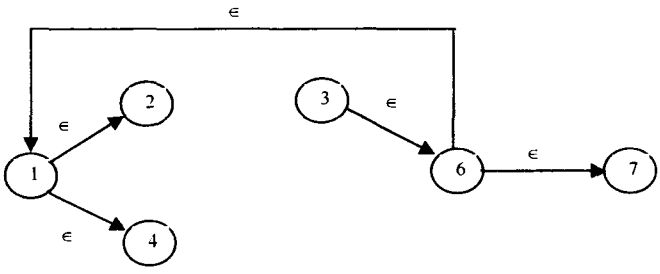
با توجه به شکل ۲-۲۶ حالات 1,2,4,7 از حالت 0 و با انتقال ∈ قابل دستیابی است. علاوه بر حالات 1,2,4,7 حالت 0 نیز باید لحاظ شود.

∈_closure(0)={0,1,2,4,7}



شکل ۲-۲۶ ∈_closure(0)

همچنین با توجه به شکل ۲-۲۷ حالات 6,7,1,2,4 از حالت 3 و با انتقال ∈ قابل دستیابی است. علاوه بر این حالات، حالت 3 نیز باید لحاظ شود.



شکل ۲-۲۷ ∈_closure(3)

∈_closure(3)={3.6.7.1.2.4}

همانطور که شکل ۲-۲۸ نشان می‌دهد از حالت 7 و با انتقال ∈ حالت مقصدی وجود ندارد در نتیجه تنها جواب همان حالت ۷ است.

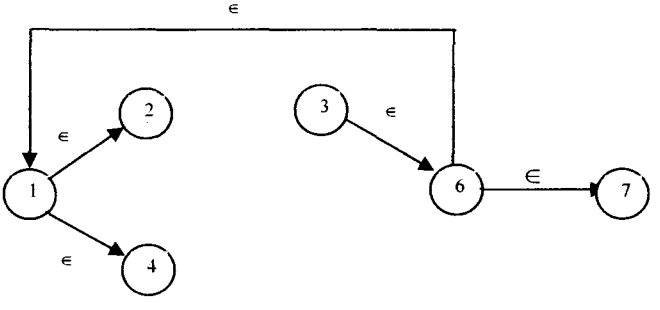
∈_closure(7)={7}

شکل ۲-۲۸ ∈_closure(7)

∈_closure(T): مجموعه حالاتی از NFA که از یکی از حالات درون T و با تبدیل ∈ قابل دسترسی است. در این تعریف T مجموعه‌ای از حالات است.

مثال۲-۱۶ با توجه به NFA شکل ۲-۲۵ موارد ذیل را محاسبه کنید.

A={6,3}
∈_closure(A)=?
A={1,4,8}
∈_closure(A)=?



شکل ۲-۲۹ ∈_closure({6,3})

A={6,3}
∈_closure(A)= ∈_closure(6) U ∈_closure(3)={6,7,1,2,4} U {3,6,7,1,2,4} = {6,7,1,2,3,6,7,4}
∈_closure({1,4,8})= ∈_closure(1) U ∈_closure(4) U ∈_closure(8)={1,2,4} U {4} U {8}={1,2,4,8}

شکل ۲-۳۰ (∈_closure({1,4,8})

move(T,a) مجموعه‌ای از حالات NFA است که از حالت s درون مجموعه T و نماد a قابل دسترسی است.

مثال۲-۱۷ با توجه به NFA شکل ۲-۲۵ موارد ذیل را محاسبه کنید.

move({2,4,7},a)=?
move({2,4,7},b)=?
move({2,4,7},c)=?
move({2,4,7},a)= move(2,a) U move (4,a) U move (7,a)={3} U {} U {}={3}
move({2,4,7},b)= move(2,b) U move (4,b) U move (7,b)={} U {5} U {}={5}
move({2,4,7},b)= move(2,c) U move (4,c) U move (7,c)={} U {5} U {}={5}

این توابع را می‌توان ترکیب کرد. به عنوان مثال :

∈_closure(move({1,2},a))= ∈_closure(move(1,a) U move(2,a)) = ∈_closure({} U{3}) = ∈_closure({3}) ={3,6,7,1,2,4}

قبلا نشان دادیم که می‌توان DFA را بوسیله یک جدول نشان داد. الگوریتم ذیل نحوه استخراج این جدول را از NFA نشان می‌دهد. نام این جدول را DTrans می‌نامیم.

با توجه به ∈_closure(s) و move ، الگوریتمی ارائه می‌دهیم که NFA را به DFA تبدیل کند.

۱- ∈_closure(s) را محاسبه می‌کنیم (s حالت شروع NFA است). به مجموعه ∈_closure(s) یک نام اختصاص می‌دهیم (مانند A) و به DTrans اضافه می‌کنیم. این مجموعه حالت شروع DFA است.

۲- یک حالت علامت نخورده درون Dtrans را یافته (این حالت را در مراحل بعدی T می‌نامیم) و مراحل ذیل را روی آن اعمال می‌کنیم(اولین بار که این مرحله اجرا می‌شود تنها حالت علامت نخورده حالت بدست آمده از مرحله ۱ الگوریتم است). اگر چنین حالتی وجود ندارد الگوریتم پایان می‌یابد.

۳- حالت T علامت می‌زنیم.

٤- برای هر نماد الفبای زبان (این نماد را در مراحل بعدی a می‌نامیم) مراحل ذیل را تکرار می‌کنیم.

٤-١- مجموعه ∈_closure((move(T,a)) را محاسبه می‌کنیم(این مجموعه را در مراحل بعدی الگوریتم U می‌نامیم). اگر این مجموعه با مجموعه‌هایی که قبلا محاسبه شده و در DTrans موجود است یکسان نباشد، نام جدیدی به آن اختصاص داده و این مجموعه را به DTrans اضافه می‌کنیم (دقت کنید که این حالت علامت نخورده است).

٤-٢- حالت بعدی T به ازای ورودی a است. در نتیجه در DTrans بخش ذیل را اضافه می‌کنیم.

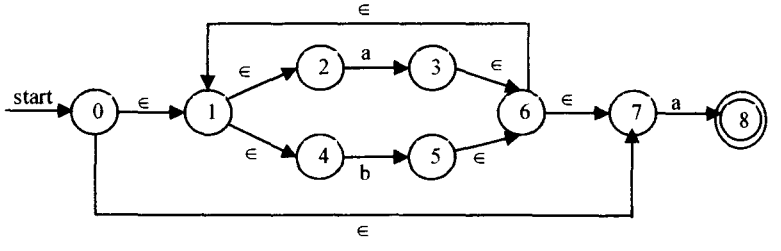
```
DTrans[T,a]=U
```

مراحل ذکر شده را می‌توان به صورت الگوریتم ذیل نیز ارائه داد.

```
initially, ∈_closure(s0) is the only state in DTrans and it is unmarked
while there is an unmarked  state T in Dtrans do
begin
        ،nark T;
        for each input symbol a do
        begin
                U:= ∈_closure(move(T,a));
                if U is not in Dtrans then
                        add  U as an unmarked state to Dtrans
                DTrans[T,a]:=U
        end
end
```

این مجموعه را که شامل یکی از حالات پذیرش NFA باشد حالت پذیرش DFA محسوب می‌شود.

مثال٢-١٨ NFAی ذیل را به DFA تبدیل کنید.



شکل ٢-٣١ NFA مربوط به (a|b)*a

با توجه به NFA نمادها، bوa است. الگوریتم ذکر شده را قدم به قدم به صورت ذیل دنبال می‌کنیم:

- با توجه به مرحله ۱ الگوریتم و با توجه به اینکه حالت شروع NFA حالت 0 است، حالت شروع DFA به صورت ذیل محاسبه می‌شود.

∈_closure(0)={0,1,2,4,7}

مجموعه جواب را A نامیده و به DTrans اضافه می‌کنیم. در این مرحله A تنها حالت علامت نخورده است.

جدول ۲-۱۰ اضافه شدن A به dtrans

	a	b
A		

- با توجه به مرحله ۲ الگوریتم، یک حالت علامت نخورده در DTrans را می‌یابیم. در این مرحله تنها حالت علامت نخورده حالت A است. حالت A را در نظر گرفته و مراحل ذیل را روی آن اعمال می‌کنیم. دقت کنید که A مجموعه حالات {0,1,2,4,7} است.

- با توجه به مرحله ۳ الگوریتم حالتA را علامت می‌زنیم.

جدول ۲-۱۱ زدن علامت برای A

	a	b
A√		

- با توجه به مرحله ٤ الگوریتم به ازای هر یک از نمادهای a,b و مجموعه A، مراحل ذیل را اجرا می‌کنیم.

- با توجه به مرحله ٤-۱ الگوریتم مجموعه U:= ∈_closure(move(A,a)) را محاسبه می‌کنیم.

move(A,a)= move({0,1,2,4,7},a)= move(0,a) U move(1,a) U move(2,a) move(4,a) U move(7,a) ={}U{}U{3}U{} U {8}={3,8}

U:= ∈_closure(move(A,a))= ∈_closure(3)= {3,6,7,1,2,4,8}

از آنجاییکه مجموعه به دست آمده با مجموعه های قبلی(یعنی مجموعه A)مساوی نیست در نتیجه یک مجموعه جدید است.یک نام جدید مانند B به آن اختصاص می‌دهیم و سپس آن را به DTrans اضافه می‌کنیم. در این حالت B تنها حالت علامت نخورده است.

جدول ۲-۱۲ اضافه شدن B به dtrans

	a	b
A√ B		

- با توجه به مرحله ٤-۲ الگوریتم DTrans[A,a]=B.

جدول ۲-۱۳ درج B برای A

	a	b
A√ B	B	

- با توجه به مرحله ٤-۱ الگوریتم U:= ∈_closure(move(A,b)) را محاسبه می‌کنیم.

move(A,b)= move({0,1,2,4,7},b)= move(0,b) U move(1,b) U move(2,b) U move(4,b) U move(7,b)={}U {}U {}U {}U{5}U{}={5}

U:= ∈_closure(move(A,b))= ∈_closure(5)= {5,6,7,1,2,4}

از آنجاییکه مجموعه به دست آمده با مجموعه های قبلی (یعنی B و A) یکسان نیست، در نتیجه یک مجموعه جدید است. یک نام جدید مانند C به آن اختصاص می‌دهیم و آنرا به DTrans اضافه می‌کنیم.

جدول ۲-۱۴ اضافه شدن C به dtrans

	a	b
A√ B C	B	

- با توجه به مرحله ٤-۲ الگوریتم DTrans[A,b]=C

جدول ۲-۱۵ درج C برای A

	a	b
A√ B C	B	C

با توجه به مرحله ۲ الگوریتم، یک حالت علامت نخورده در DTrans را می‌یابیم. در این مرحله حالت علامت نخورده حالت B است. حالت B را در نظر گرفته و مراحل ذیل را روی آن اعمال می‌کنیم. دقت کنید که B مجموعه حالات {3,6,7,1,2,4,8} است.

- با توجه به مرحله ۳ الگوریتم حالت B را علامت می‌زنیم.

جدول ۲-۱۶ علامت زدن A

	a	b
A√ B√ C	B	C

- با توجه به مرحله ٤ الگوریتم به ازای هر یک از نمادهای a,b و مجموعه B، مراحل ذیل را اجرا می‌کنیم.

- با توجه به مرحله ٤-١ الگوریتم مجموعه U:=ϵ_closure(move(B,a)) را محاسبه می‌کنیم.

move(B,a)= move({3,6,7,1,2,4,8},a)= move(3,a) U move(6,a) U move(7,a) U move(1,a) U move(2,a) U move(4,a) U move(8,a) ={}U{}U{8}U{} U {3}U {} U {}={3,8}
U:= ϵ_closure(move(B,a))= ϵ_closure({3,8})= {3,6,7,1,2,4,8}

مجموعه {3,6,7,1,2,4,8}، مجموعه جدیدی نیست. این مجموعه، با مجموعه B برابر است. در نتیجه نیازی به نام جدید نمی‌باشد.

- با توجه به مرحله ٤-٢ الگوریتم DTrans[B,a]=B.

جدول ۲-۱۷ درج B برای B

	a	b
A√	B	C
B√	B	
C		

- با توجه به مرحله ٤-١ الگوریتم U:= ϵ_closure(move(B,b)) را محاسبه می‌کنیم.

move(B,b)= move({3,6,7,1,2,4,8},b)= move(3,b) U move(6,b) U move(7,b) U move(1,b) U move(2,b) U move(4,b) U move(8,b) ={}U {}U {}U {}U{}U{5} U{}={5}
U:= ϵ_clousre(move(B,b))= ϵ_clousre(5)= {5,6,7,1,2,4}

مجموعه {5,6,7,1,2,4}، مجموعه جدیدی نیست. این مجموعه، با مجموعه C برابر است. در نتیجه نیازی به نام جدید نمی‌باشد.

- با توجه به مرحله ٤-٢ الگوریتم DTrans[B,b]=C

جدول ۲-۱۸ درج C برای B

	a	b
A√	B	C
B√	B	C
C		

- با توجه به مرحله ۲ الگوریتم، یک حالت علامت نخورده در DTrans می‌یابیم. در این مرحله حالت علامت نخورده حالت C است. حالت C را در نظر گرفته و مراحل ذیل را روی آن اعمال می‌کنیم. دقت کنید که C مجموعه حالات {5,6,7,1,2,4} است.

- با توجه به مرحله ۳ الگوریتم حالتC را علامت می‌زنیم.

جدول ۲-۱۹ درج علامت برای C

	a	b
A√	B	C
B√	B	C
C√		

- با توجه به مرحله ٤ الگوریتم به ازای هر یک از نمادهای a,b و مجموعه C، مراحل ذیل را اجرا می‌کنیم.

- با توجه به مرحله ٤-١ الگوریتم مجموعه U:=ϵ_closure(move(C,a)) را محاسبه می‌کنیم.

move(C,a)= move({5,6,7,1,2,4},a)= move(5,a) U move(6,a) U move(7,a) U move(1,a) U move(2,a) U move(4,a) ={}U{}U{8}U{} U {3}U {} ={3,8}
U:= ϵ_closure(move(C,a))= ϵ_closure({3,8})= {3,6,7,1,2,4,8}

مجموعه {3,6,7,1,2,4,8}، مجموعه جدیدی نیست. این مجموعه، با مجموعه B برابر است. در نتیجه نیازی به نام جدید نمی‌باشد.

- با توجه به مرحله ٤-٢ الگوریتم DTrans[C,a]=B.

جدول ۲-۲۰ درج B برای C

	a	b
A√	B	C
B√	B	C
C√	B	

- با توجه به مرحله ٤-١ الگوریتم U:= ϵ_closure(move(C,b)) را محاسبه می‌کنیم.

move(C,b)= move({5,6,7,1,2,4},b)= move(5,b) U move(6,b) U move(7,b) U move(1,b) U move(2,b) U move(4,b) ={}U {}U {}U {}U{}U{5}={5}
U:= ϵ_clousre(move(B,b))= ϵ_clousre(5)= {5,6,7,1,2,4}

مجموعه {5,6,7,1,2,4}، مجموعه جدیدی نیست. این مجموعه، با مجموعه C برابر است. در نتیجه نیازی به نام جدید نمی‌باشد.

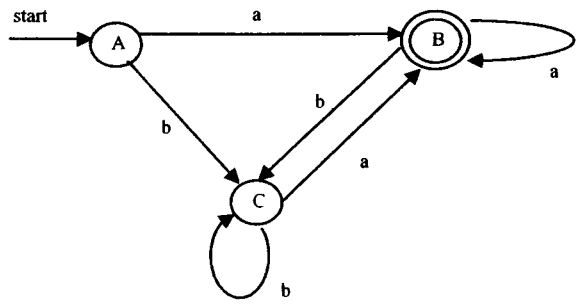
- با توجه به مرحله ٤-٢ الگوریتم DTrans[C,b]=C

جدول ۲-۲۱ درج C برای C

	a	b
A√	B	C
B√	B	C
C√	B	C

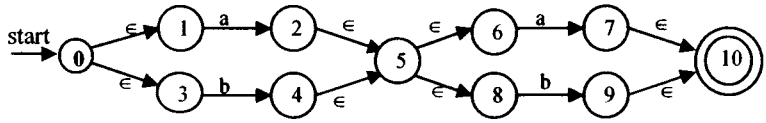
در این مرحله، حالت علامت نخورده ای وجود ندارد بنابراین الگوریتم پایان می‌یابد.

چون مجموعه B شامل حالت پذیرش NFA (یعنی حالت ۸) است، در نتیجه B حالت پذیرش DFA ی به دست آمده است. با توجه به جدول تغییر حالت به دست آمده، DFA حاصل به صورت ذیل خواهد بود.



شکل ۲-۳۲ DFA ایجاد شده از NFA شکل ۲-۳۱

مثال۲-۱۹ NFAی ذیل را به DFA تبدیل کنید.



شکل ۲-۳۳ NFA مربوط به (a|b)(a|b)

ϵ_closure(0)={0,1,3}

مجموعه {0,1,3} را A می‌نامیم.

	a	b
A		

ϵ_closure(move(A,a)) = ϵ_closure(move({0,1,3},a) = ϵ_closure((move(0,a) U move(1,a) U move(3,a)) = ϵ_closure({}U{2} U {}) = ϵ_closure ({2})= {2,5,6,8}

مجموعه {2,5,6,8} را B می‌نامیم.

	a	b
A	B	
B		

ϵ_closure(move(A,b)) = ϵ_closure(move({0,1,3},b) = ϵ_closure((move(0,b) U move(1,b) U move(3,b)) = ϵ_closure({}U{} U {4}) = ϵ_closure ({4})= {4,5,6,8}

مجموعه {4,5,6,8} را C می‌نامیم.

	a	b
A	B	C
B		
C		

ϵ_closure(move(B,a)) = ϵ_closure(move({2,5,6,8},a) = ϵ_closure((move(2,a) U move(5,a) U move(6,a) U move(8,a)) = ϵ_closure({}U{}U {7} U {}) =

مجموعه {7,10} را D می‌نامیم.

	a	b
A	B	C
B	D	
C		
D		

ϵ_closure(move(B,b)) = ϵ_closure(move({2,5,6,8},b) = ϵ_closure((move(2,b) U move(5,b) U move(6,b) U move(8,b)) = ϵ_closure({}U{}U {} U {9}) =

مجموعه {9,10} را E می‌نامیم.

	a	b
A	B	C
B	D	E
C		
D		
E		

ϵ_closure(move(C,a)) = ϵ_closure(move({4,5,6,8},a) = ϵ_closure((move(4,a) U move(5,a) U move(6,a) U move(8,a)) = ϵ_closure({}U{}U {7} U {}) = ϵ_closure({7})= {7,10}

مجموعه {7,10} همان مجموعه D است که قبلا به دست آمد.

	a	b
A	B	C
B	D	E
C	D	
D		
E		

ϵ_closure(move(C,b)) = ϵ_closure(move({4,5,6,8},b) = ϵ_closure((move(4,b) U move(5,b) U move(6,b) U move(8,b)) = ϵ_closure({} U {} U {} U {9}) = ϵ_closure({9})= {9,10}

مجموعه {9,10} همان مجموعه E است که قبلا به دست آمد.

	a	b
A	B	C
B	D	E
C	D	E
D		
E		

∈_closure(move(D,a)) = ∈_closure(move({7,10},a) = ∈_closure((move(7,a) U move(10,a)) = ∈_closure({}U{}) = ∈_closure({})= {}

	a	b
A	B	C
B	D	E
C	D	E
D		
E		

∈_closure(move(D,b)) = ∈_closure(move({7,10},b) = ∈_closure((move(7,b) U move(10,b)) = ∈_closure({}U{}) = ∈_closure({})= {}

	a	b
A	B	C
B	D	E
C	D	E
D		
E		

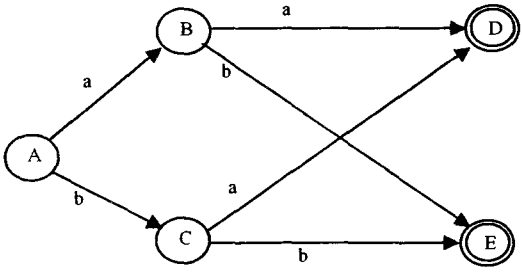
∈_closure(move(E,a)) = ∈_closure(move({8,10},a) = ∈_closure((move(8,a) U move(10,a)) = ∈_closure({}U{}) = ∈_closure({})= {}

	a	b
A	B	C
B	D	E
C	D	E
D		
E		

∈_closure(move(E,b)) = ∈_closure(move({8,10},b) = ∈_closure((move(8,b) U move(10,b)) = ∈_closure({}U{}) = ∈_closure({})= {}

	a	b
A	B	C
B	D	E
C	D	E
D		
E		

مجموعه های {9,10}=D و {7,10}=E شامل حالت نهایی 10 هستند بنابراین حالات D و E حالت نهایی DFA هستند.



شکل ۲-۳۴ DFA مربوط به NFA شکل ۲-۳۳

با توجه به اینکه تعداد حالات NFA تولید شده به روش تامپسون متناسب با |r| است و با توجه به اینکه هر حالت DFA تولید شده از NFA زیر مجموعه ای از حالات NFA است در نتیجه تعداد حالات DFA به دست آمده حداکثر $2^{|r|}$ است[1]، بنابراین پیچیدگی فضای لازم برای ذخیره DFA، $O(2^{|r|})$ است.

زمان لازم برای تعیین پذیرش یا عدم پذیرش رشتهx توسط DFA، |x| است (|x| طول رشته x است) زیرا در DFA با هر تغییر حالت یک نماد از رشته x مصرف می‌شود بنابراین پیچیدگی زمان لازم برای تعیین پذیرش یا عدم پذیرش رشته x توسط DFA، O(|x|) است.

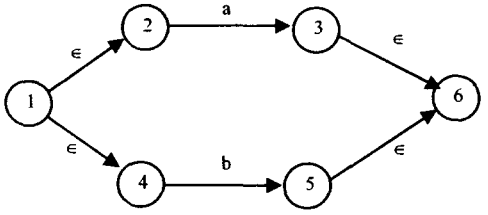
۲-۱۳ ساخت مستقیم DFA از عبارت باقاعده

در بخش قبل روشی برای ایجاد یک DFA برای عبارت باقاعده ارائه گردید. روش ذکر شده مستلزم ایجاد یک NFA و سپس ایجاد DFA از NFA بود. در این قسمت روشی برای ایجاد یک DFA برای عبارت باقاعده بدون ساخت NFA تشریح می‌کنیم.

---

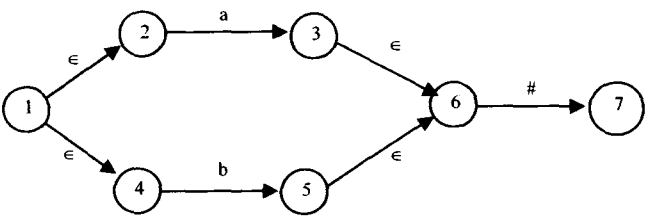
۱. زیرا تعداد زیر مجموعه های ، مجموعه ای با r عضو، 2^r است.

در ساخت NFA از عبارت باقاعده r برای هر نماد a در r یک حالت با لبه خروجی با برچسب a ایجاد می‌گردد و لبه های دیگر NFA برچسب ∈ دارند. به عنوان مثال NFA عبارت باقاعده r=a|b در ذیل نشان داده شده است.



شکل ۲-۳۵ NFA مربوط به (a|b)

در این NFA فقط دو لبه با برچسب a و b وجود دارد که متناظر با a و b در عبارت باقاعده هستند و بقیه لبه ها برچسب ∈ دارند. در ساخت DFA از NFA برای هر نماد a از عبارت باقاعده ∈_closure(move(s,a)) محاسبه می‌گردد اگر همه لبه های خروجی از حالت s برچسب ∈ داشته باشند move(s,a) تهی و در نتیجه ∈_closure(move(s,a)) نیز تهی می‌گردد. اگر ∈_closure(move(s,a)) تهی نباشد یکی از حالات DFA خواهد شد. در نتیجه در ساخت DFA از NFA فقط حالاتی که خروجی با برچسب غیر∈ داشته باشند باعث تولید حالات DFA می‌شوند. حالاتی که خروجی غیر∈ دارند حالاتی هستند که متناظر با نمادهای عبارت باقاعده هستند به عنوان مثال در NFA مربوط به (a|b) لبه خروجی حالت 2 متناظر با a و لبه خروجی حالت 4 متناظر با b در عبارت باقاعده هستند.این مطلب در ساخت مستقیم DFA از عبارت باقاعده نیز به کار می‌رود، تفاوت اصلی در حالت پذیرش است. حالت پذیرش NFA خروجی ندارد در نتیجه اگر s حالت پذیرش باشد، ∈_closure(move(s,a)) آن تهی می‌گردد. برای اینکه حالت پذیرش نیز دارای برچسب غیر تهی باشد عبارت باقاعده r را به #r تبدیل می‌کنیم. به عنوان مثال (a|b) به عبارت باقاعده افزوده #(a|b) تبدیل می‌گردد. NFA برای #(a|b) در شکل ذیل نشان داده شده است. در NFA افزوده حالت 6 که حالت پذیرش است ، خروجی # دارد.

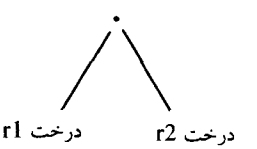


شکل ۲-۳۶ NFA مربوط به #(a|b)

با توجه به آنچه ذکر شد، در تبدیل مستقیم عبارت باقاعده r به DFA، r را به #r تبدیل می‌کنیم. پس از تبدیل r به #r ، درخت دستور را برای #r ایجاد کرده وسپس با استفاده از آن DFA را ایجاد می‌کنیم. عبارت باقاعده را می‌توان به وسیله درخت دستور نمایش داد. برای تبدیل عبارت باقاعده به درخت دستور از قوانین ذیل استفاده می‌کنیم.

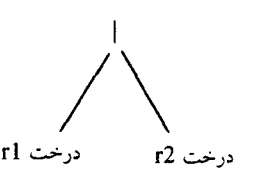
۱- هر نماد یا ϵ در عبارت باقاعده r به برگ تبدیل می‌شود.

۲- عملگر الحاق به گره داخلی تبدیل می‌شود. به عنوان مثال درخت دستور r1.r2 به صورت ذیل ایجاد می گردد.



شکل ۲-۳۷ تبدیل عملگر الحاق به گره داخلی

۳- عملگر or به گره داخلی تبدیل می‌شود. به عنوان مثال r1|r2 به صورت ذیل به درخت دستور تبدیل می شود.



شکل ۲-۳۸ تبدیل عملگر or به گره داخلی

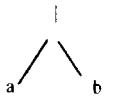
٤- عملگر * به گره داخلی تبدیل می‌شود. به عنوان مثال r* به صورت ذیل به درخت دستور تبدیل می شود.

شکل ۲-۳۹ تبدیل عملگر * به گره داخلی

مثال۲-۲۰ r=(a|b)*abb را به درخت دستور تبدیل می‌کنیم. ابتدا عبارت منظم r را به عبارت باقاعده r# تبدیل می‌کنیم، بنابراین:

r#=(a|b)*abb#

- با توجه به قانون ۳ درخت a|b را می‌سازیم.

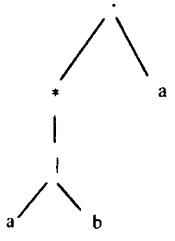


شکل ۲-۴۰ تبدیل a|b به گره داخلی

- با توجه به قانون ٤ درخت (a|b)* را می‌سازیم.

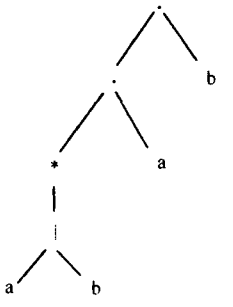
شکل ۲-٤۱ تبدیل (a|b)* به گره داخلی

- با توجه به قانون ۲ و مرحله قبل درخت ((a|b)*).a را می‌سازیم.



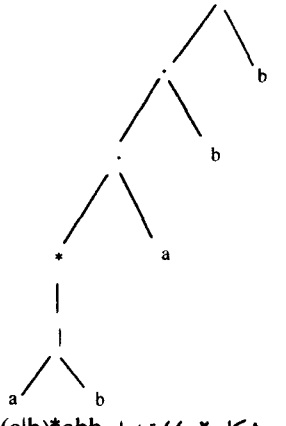
شکل ۲-٤۲ تبدیل (a|b)*a

- با توجه به قانون ۲ و مرحله قبل درخت ((a|b)*a).b را می‌سازیم.



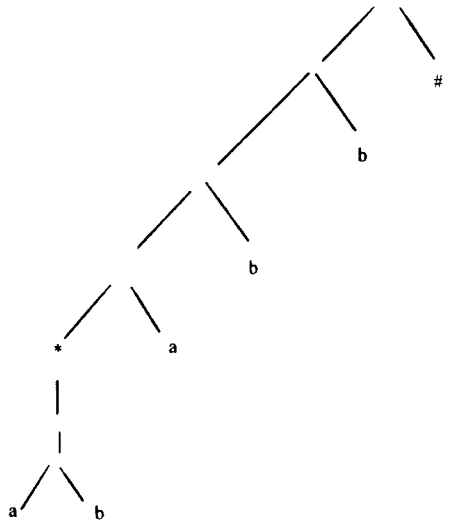
شکل ۲-۴۳ تبدیل (a|b)*ab

- با توجه به قانون ۲ و مرحله قبل درخت ((a|b)*ab)b را می‌سازیم.



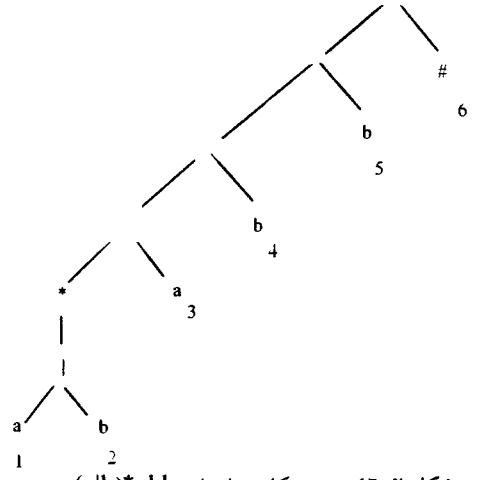
شکل ۲-۴۴ تبدیل (a|b)*abb

- با توجه به قانون ۲ و مرحله قبل درخت #(a|b)*abb را می‌سازیم.



شکل ۲-٤٥ تبدیل #(a|b)*abb

به هر نماد (به جز ∈)، عددی متناسب با مکان نماد در عبارت باقاعده اختصاص داده می‌شود. این عدد، مکان نماد یا مکان برگ نامیده می‌شود. به درخت دستور ذیل دقت کنید.



شکل ۲-٤٦ تعیین مکان نمادهای (a|b)*abb

برای ساخت DFA از درخت دستور r# از چندین تابع استفاده می‌کنیم. اگر n مکانی در درخت دستور باشد آنگاه :

nullable(n): این تابع مشخص می‌کند آیا زیر درخت n می‌تواند ∈ را تولید کند یا خیر. اگر بتوان ∈ را از زیر درخت دستور n تولید کرد، مقدار بازگشتی تابع true و در غیر این صورت false است.

با توجه به تعریف nullable قوانین ذیل برقرار هستند.

۱- اگر n برگی با برچسب ∈ باشد آنگاه nullable(n)=True است.

اگر r= a باشد آنگاه:

nullable(n)=False

۲- اگر r یک گره ستاره دار باشد، آنگاه Nullable(n)=True، زیرا، عملگر* می‌تواند ∈ تولید کند.

مثال۲-۲۱ r=a*

در این مثال r قادر به تولید ∈ خواهد بود. در نتیجه:

Nullable(n)=True

مثال۲-۲۲ r=(a|b)*

در این مثال r قادر به تولید ∈ است. در نتیجه:

Nullable(n)=True

مثال۲-۲۳ r=ab*

Nullable(n)=False

زیرا در این مثال r قادر به تولید ∈ نیست. زیرا هر رشته تولید شده توسط r حداقل یک a خواهد داشت.

firstpos(n): این تابع مجموعه مکان نمادهایی را بر می‌گرداند که منطبق بر اولین نماد رشته‌های تولیدی از عبارت باقاعده هستند.

مثال۲-۲٤ firstpos را برای عبارت باقاعده ab* مشخص نمایید.

رشته های تولید شده توسط ab* عبارتند از:

a , ab , abb , abbb , abbbb, ...

همانطور که ملاحظه می‌شود اول تمام رشته‌ها، a است و این a از اولین نماد در عبارت باقاعده ab* بدست آمده است لذا firstpos={1}

مثال۲-۲۵ firstpos را برای عبارت باقاعده (a|b)*c مشخص نمایید.

رشته های تولید شده توسط (a|b)*c عبارتند از:

c, ac, abc , abbc , abbbc , abbbbc, ..., bc, baac,baaaac,...,ababc, abbaac, aabbaabbc, ...

همانطور که ملاحظه می‌شود اول تمام رشته های تولیدی a,b یا c است که از اولین و دومین و سومین نماد در عبارت با قاعده **(a|b)*c** بدست آمده است لذا firstpos={1,2,3}

مثال۲-۲۶ firstpos را برای عبارت با قاعده (a|b)+c مشخص نمایید.

رشته های تولید شده توسط (a|b)+c عبارتند از:

ac, abc , abbc , abbbc , abbbbc, ..., bc, baac,baaaac,...,ababc, abbaac, aabbaabbc, ...

همانطور که ملا حظه می‌شود اول تمام رشته های تولیدی b ویا a می‌باشد و این a یا b از اولین و دومین نماد در عبارت باقاعده **(a|b)**+c بدست آمده است بنابراین firstpos={1,2}

مثال۲-۲۷ firstpos را برای عبارت با قاعده (a|b)+a مشخص نمایید.

رشته های تولید شده توسط (a|b)+a عبارتند از:

aa, aba , abba , abbba , abbbba, ..., bc, baac,baaaaa,...,ababa, abbaaa, aabbaabba, ...

همانطور که ملاحظه می‌شود اول تمام رشته های تولیدیb یا a است و این a یا b از اولین و دومین نماد در عبارت با قاعده **(a|b)**+a بدست آمده است لذا firstpos={1,2}. در اینجا باید دقت نمود که نماد a در مکان اول عبارت با قاعده با نماد a واقع در مکان سوم عبارت با قاعده متفاوت است.

lastpos(n): این تابع مجموعه مکان نمادهایی را بر می‌گرداند که منطبق بر آخرین نماد رشته های تولیدی از عبارت باقاعده هستند.

مثال۲-۲۸ lastpos را برای عبارت با قاعده *ab مشخص نمایید.

رشته های تولید شده توسط *ab عبارتند از:

a , ab , abb , abbb , abbbb, ...

همانطور که ملاحظه می‌شود آخر تمام رشته ها b و یا a می‌باشد و این b و یا a از اولین و دومین نماد در عبارت باقاعده **ab*** بدست آمده است لذا lastpos={1,2}

مثال۲-۲۹ lastpos را برای عبارت با قاعده (a|b)*c مشخص نمایید.

رشته های تولید شده توسط (a|b)*c عبارتند از:

c, ac, abc , abbc , abbbc , abbbbc, ..., bc, baac,baaaac,...,ababc, abbaac, aabbaabbc, ...

همانطور که ملا حظه می‌شود آخرین نماد تمام رشته ها تولیدی c می‌باشد و c از سومین نماد در عبارت باقاعده **(a|b)*c** بدست آمده است لذا lastpos={3}

مثال۲-۳۰ lastpos را برای عبارت با قاعده (a|b)+c مشخص نمایید.

رشته های تولید شده توسط (a|b)+c عبارتند از:

ac, abc , abbc , abbbc , abbbbc, ..., bc, baac,baaaac,...,ababc, abbaac, aabbaabbc, ...

همانطور که ملا حظه می‌شود آخر تمام رشته های تولیدیc است و این c از سومین نماد در عبارت با قاعده (a|b)+c بدست آمده است لذا lastpos={3}

مثال۲-۳۱ lastpos را برای عبارت با قاعده (a|b)+a مشخص نمایید.

رشته های تولید شده توسط (a|b)+a عبارتند از:

aa, aba , abba , abbba , abbbba, ..., baa, baaaa,baaaaa,...,ababa, abbaaa, aabbaabba, ...

همانطور که ملاحظه می‌شود آخر تمام رشته‌ها تولیدی a است و این a از سومین نماد در عبارت باقاعده (a|b)+a بدست آمده است لذا lastpos={3}. دقت کنید که نماد a در اول عبارت باقاعده با نماد a واقع در مکان سوم عبارت باقاعده متفاوت است.

بین این توابع روابط ذیل برقرار است:

۱- اگر r یک or-node با سمت چپ r1 و سمت راست r2 باشدآنگاه:

```
firstpos(r)=firstpos(r1) U firstpos(r2)
lastpos(r)= lastpos(r1) U lastpos(r2)
nullable(r)= nullable(r1) or nullable(r2)
```

۲- اگر r یک cat-node با سمت چپ r1 و سمت راست r2 باشد آنگاه:

```
if (nullable(r1)=TRUE)
        firstpos(r)=firstpos(r1) U firstpos(r2)
else
        firstpos(r)=firstpos(r1)


if (nullable(r2)=TRUE)
        lastpos(r)= lastpos(r1) U lastpos(r2)
else
        lastpos(r)=lastpos(r2)

nullable(r)= nullable(r1) and nullable(r2)
```

۳- اگر r یک star-node از r1 باشد، آنگاه:

```
firstpos(r)=firstpos(r1)
lastpos(r)= lastpos(r1)
nullable(r)= TRUE
```

followpos(i): این تابع مجموعه مکان نمادهایی را بر می‌گرداند که پس از مکان i قرار می‌گیرند.

مثال۲-۳۲ followpos(1): را برای عبارت با قاعده *ab مشخص نمایید.

در *ab، نماد a در مکان یک و b در مکان دو قرار دارد. اگر نمادی از رشته تولیدی منطبق بر a در مکان یک عبارت باقاعده باشد، نماد بعد از آن منطبق بر b در عبارت باقاعده است و مکان b نیز ۲ است. بنابراین followpos(1)={2} می‌شود.

مثال۲-۳۳ followpos(1) را برای عبارت با قاعده *(ab) مشخص نمایید.

در این عبارت a در مکان یک و b در مکان دو قرار دارد. اگر رشته‌هایی که توسط این عبارت باقاعده تولید می‌شوند را در نظر بگیریم مشخص می‌شود اگر نمادی منطبق بر a تولید شود نماد بعد از آن می‌تواند منطبق بر a و یا b در عبارت باقاعده باشد. در این صورت followpos(1)={1,2} است. به همین دلیل followpos(2)={1,2} زیرا اگر در رشته‌های تولیدی نمادی بر b منطبق گردد نماد بعدی می‌تواند بر a و یا b منطبق شود.

مثال۲-۳٤ followpos(1) را برای عبارت با قاعده (a|b)*c مشخص نمایید.

در این عبارت a در مکان یک و b در مکان دو و c در مکان سه قرار دارد. اگر رشته‌هایی که توسط این عبارت تولید می‌شوند را در نظر بگیریم مشخص می‌شود اگر نمادی در یک رشته منطبق برa تولید شود نماد بعد از آن می‌تواند منطبق بر b یا a و یا c در عبارت با قاعده باشد. در این صورت followpos(1)={1,2,3} می‌شود.

برای محاسبه followpos(i) می‌توان از قوانین ذیل نیز استفاده نمود.

۱- اگر n یک cat-node با فرزند سمت چپ c1 و فرزند سمت راست c2 باشد و i مکانی در مجموعه lastpos(c1) باشد، تمام مکانهای مجموعه firstpos(c2) در followpos(i) قرار دارند.

۲- اگر n یک star-node و i مکانی در مجموعه lastpos(n) باشد، تمام مکانهای مجموعه firstpos(n) در followpos(i) قرار دارند.

مثال۲-۳۵ در عبارت با قاعده (a|b|c)*(c|d) ، followpos(3) را محاسبه کنید.

چون3 در lastpos مربوط به star-node، (a|b|c)* قرار دارد، بنابراین تمام مکانهای firstpos به followpos(3) اضافه می‌شوند، بنابراین :

followpos(3)={1,2,3}

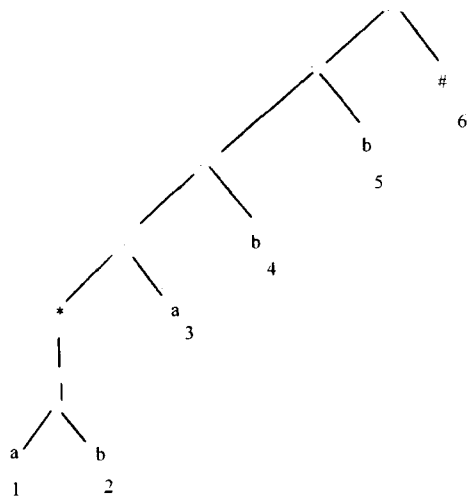
همچنین چون3 در lastpos سمت چپ مربوط به cat-node، (a|b|c)*.(c|d) قرار دارد در نتیجه مکانهای firstpos مربوط به (c|d) در followpos(3) قرار دارند. بنابراین:

followpos(3)={1,2,3,4,5}

به منظور ایجادDFA برای عبارت با قاعده r از الگوریتم ذیل استفاده می‌شود. برای اجرای این الگوریتم جدولی به نام dtrans که جدول تغییر حالات DFA است را در نظر می‌گیریم و مراحل ذیل را روی #r اجرا می‌کنیم.

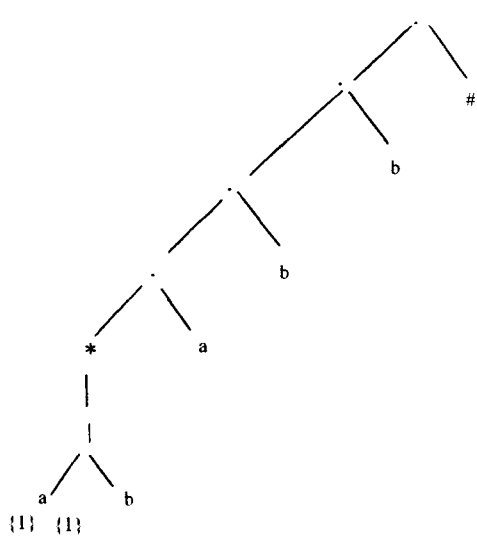
۱- درخت دستورعبارت باقاعده #r را ایجاد می‌کنیم.
۲- توابع nullable,firstpos,lastpos,followpos را در پیمایش عمقی درخت محاسبه می‌کنیم.
۳- firstpos(root) را محاسبه می‌کنیم (root ریشه در خت دستور است) و آن را به عنوان یک حالت علامت نخورده به dtrans اضافه می‌کنیم. firstpos(root) حالت شروع DFA است.
۴- اگر در dtrans حالت علامت نخورده ای وجود دارد آن را انتخاب می‌کنیم، در این الگوریتم این حالت را T می‌نامیم. اگر حالت علامت نخورده وجود ندارد الگوریتم تمام می‌شود.
۵- حالت انتخاب شده T را علامت می‌زنیم.
۶- برای هر نماد a مراحل ذیل را اجرا می‌کنیم.
۶-۱- مجموعه مکانهایی درT که نماد متناظر در عبارت با قاعده a است را محاسبه کرده و آن را p می‌نامیم.
۶-۲- مجموعه followpos(p) را محاسبه می‌کنیم مجموعه به دست آمده راU می‌نامیم.
۶-۳- U را به جدول Dtrans اضافه می‌کنیم.
۶-۴- dtarns[T,a]=U
۷- به مرحله ٤ برگرد.
۸- حالتی که شامل مکان متناظر با # است حالت پذیرش DFA است.

برای درک بهتر الگوریتم یک مثال ارائه می‌کنیم. می‌خواهیم عبارت باقاعده r=(a|b)*abb را به DFA تبدیل کنیم. ابتدا علامت # را به آخر عبارت باقاعده اضافه می‌کنیم درنتیجه #(a|b)*abb به دست می‌آید، با توجه به مرحله اول الگوریتم برای این #(a|b)*abb درخت نحو به صورت ذیل ایجاد می‌شود.



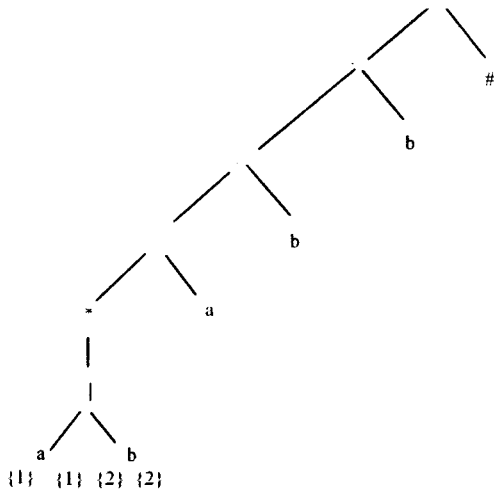
شکل ۲-۴۷ تبدیل (a|b)*abb به درخت دستور

با توجه به مرحله دوم الگوریتم firstpos و lastpos را برای درخت دستور محاسبه می‌کنیم. ابتدا a در نظر می‌گیریم که در مکان 1 قرار دارد. اگر a را به تنهایی به عنوان عبارت باقاعده در نظر بگیریم {1}=(1)firstpos و {1}=(1)lastpos می‌باشد.



شکل ۲-۴۸ محاسبه firstpos و lastpos

در این شکل مجموعه سمت چپ firstpos و مجموعه سمت راست lastpos را نشان می‌دهد. بعد از مکان یک firstpos و lastpos را برای مکان دو یعنی b محاسبه می‌کنیم. اگر b را به تنهایی در نظر بگیریم firstpos(2)={2} و lastpos(2)={2} می‌باشد.

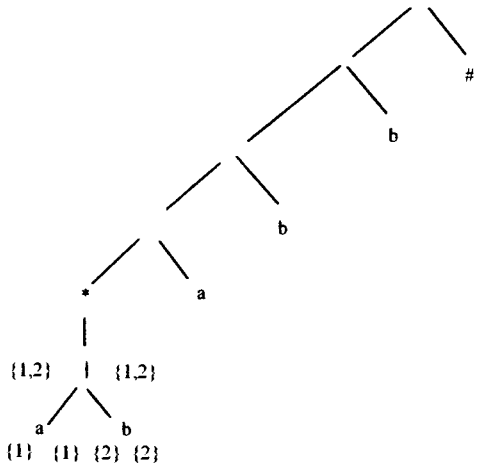


شکل ۲-۴۹ محاسبه firstpos و lastpos

گره بعدی or-node است. firstpos و lastpos آن را با استفاده از مجموعه firstpos(1) و lastpos(1) ، firstpos(2) و lastpos(2) به صورت ذیل محاسبه می‌کنیم.

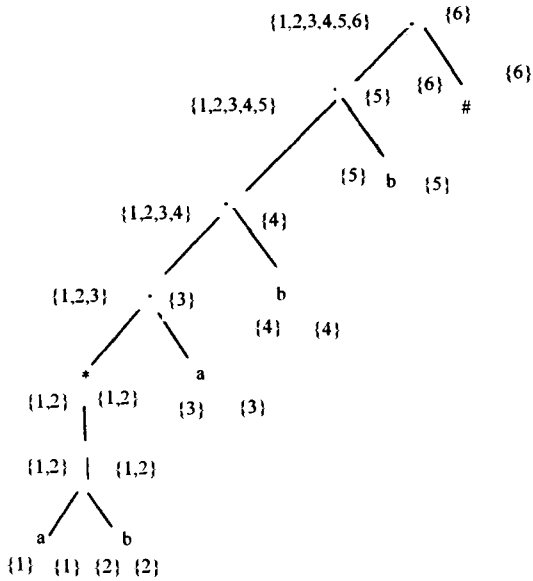
firstpos(1) U firstpos(2)={1,2}
lastpos(1) U lastpos(2)={1,2}

نتیجه در شکل ذیل ارائه شده است.



شکل ۲-۵۰ محاسبه firstpos و lastpos

بقیه موارد را هم به همین ترتیب محاسبه می‌کنیم. نتیجه در شکل ذیل نشان داده شده است.



شکل ۲-۵۱ محاسبه firstpos و lastpos

پس از محاسبه firstpos , lastpos همه نودهای درخت می‌توان با استفاده از قوانین ذکر شده، followpos را محاسبه می‌کنیم، نتیجه در شکل ذیل ارائه شده است.

جدول ۲-۲۲ محاسبه followpos

مکان	followpos
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	-

- با توجه به مرحله سوم الگوریتم، firstpos(root) را محاسبه می‌کنیم که با توجه به درخت دستور، مجموعه {1,2,3} به دست می‌آید. مجموعه {1,2,3} را A می‌نامیم، به جدول dtrans اضافه می‌کنیم. A حالت اولیه DFA است.

حالت	a	b
A		

- با توجه به مرحله چهارم الگوریتم، حالت علامت نخورده را در dtrans انتخاب می‌کنیم. در این وضعیت تنها حالت علامت نخورده A است.
- با توجه به مرحله پنجم الگوریتم، حالت انتخاب شده (A) را علامت می‌زنیم.

حالت	a	b
A√		

- با توجه به مرحله ششم، برای نماد a مراحل ذیل را انجام می‌دهیم.
- با توجه به مرحله ٦-١ الگوریتم، برای نماد a از مجموعه A مکانهایی را انتخاب می‌کنیم، که نماد متناظر آنها در عبارت با قاعدهa است در این صورت p={1,3} می‌شود. زیرا در مکانهای ۱ و ۳ در عبارت باقاعده، نماد a وجود دارد.
- با توجه به مرحله ٦-٢ الگوریتم، followpos(3) U followpos(1) را محاسبه می‌کنیم. با توجه به جدول followpos این مجموعه برابر است با {1,2,3,4}. این مجموعه با مجموعه‌های قبلی متفاوت است. در نتیجه نام دیگری مانند B به آن تخصیص می‌دهیم.
- با توجه به مرحله ٦-٣ الگوریتم، حالتB را به جدول Dtrans اضافه می‌کنیم.
- با توجه به مرحله ٦-٤ الگوریتم، dtrans[A,a]= B ،در این صورت dtrans به صورت ذیل می‌شود.

حالت	a	b
A√	B	
B		

- با توجه به مرحله ٦ الگوریتم، به ازای نمادb مراحل ذیل را انجام می‌دهیم.
-با توجه به مرحله ٦-١ الگوریتم، برای نمادb از مجموعه A={1,2,3} مکانهایی را انتخاب می‌کنیم که نماد متناظر آنها در عبارت باقاعده b است، در این صورت p={2} می‌شود. زیرا در مکان ۲، نماد b وجود دارد.
- با توجه به مرحله ٦-٢ الگوریتم، followpos(2) را محاسبه می‌کنیم. با توجه به جدول محاسبه followpos این مجموعه برابر است {1,2,3}. این مجموعه با مجموعه A برابر است لذا نام جدید لازم نیست.
- با توجه به مرحله ٦-٣ الگوریتم، dtrans[A,b]= A

در این صورت dtrans به صورت ذیل می‌شود.

حالت	a	b
A√	B	A
B		

- با توجه به مرحله چهارم الگوریتم، حالت علامت نخورده بعدی را در dtrans انتخاب می‌کنیم. در این وضعیت تنها حالت علامت نخورده B است. دقت کنید که B={1,2,3,4} است.

- با توجه به مرحله پنجم الگوریتم، حالت انتخاب شده را علامت می‌زنیم.

حالت	a	b
A√	B	A
B√		

- با توجه به مرحله ششم برای نماد a و حالت B مراحل ذیل را انجام می‌دهیم.

- با توجه به مرحله ٦-١ الگوریتم، برای نمادa از مجموعه B مکانهایی را انتخاب می‌کنیم که نماد متناظر آنها در عبارت با قاعدهa است. در این صورت p={1,3} می‌شود. زیرا در مکانهای ١ و ٣، نماد a وجود دارد.

- با توجه به مرحله ٦-٢ الگوریتم، followpos(3) U followpos(1) را محاسبه می‌کنیم. با توجه به جدول followpos این مجموعه برابر است با{1,2,3,4}. این مجموعه با مجموعه B برابر است.

- با توجه به مرحله ٦-٤ الگوریتم، dtrans[B,a]= B ،در این صورت dtrans به صورت ذیل می‌شود.

حالت	a	b
A√	B	A
B√	B	

- با توجه به مرحله ٦ الگوریتم، به ازای نمادb مراحل ذیل را انجام می‌دهیم.

- با توجه به مرحله ٦-١ الگوریتم، در این مرحله به ازای نمادb از مجموعه B مکانهایی را انتخاب می‌کنیم که نماد مطابق آنها در عبارت باقاعده b است، در این صورت p={2,4} می‌شود. زیرا در مکان ٢ و 4 نماد b قرار دارد.

- با توجه به مرحله ٦-٢ الگوریتم، followpos(2) U followpos(4) را محاسبه می‌کنیم. با توجه به جدول محاسبه followpos این مجموعه برابر است با {1,2,3,5}. این مجموعه، یک مجموعه جدید است در نتیجه نام جدیدC به آن منتسب کرده و آنرا به جدول اضافه می‌کنیم.

- با توجه به مرحله ٦-٣ الگوریتم، dtrans[B,b]= C. در این صورت جدول dtrans به صورت ذیل می‌شود.

حالت	a	b
A√	B	A
B√	B	C
C		

- با توجه به مرحله چهارم الگوریتم، حالت علامت نخورده بعدی را در dtrans انتخاب می‌کنیم. در این وضعیت تنها حالت علامت نخورده C است. دقت کنید که C={1,2,3,5} است.

- با توجه به مرحله پنجم الگوریتم، حالت انتخاب شده را علامت می‌زنیم.

حالت	a	b
A√	B	A
B√	B	C
C√		

- با توجه به مرحله ششم به ازای نماد a و حالتC مراحل ذیل را انجام می‌دهیم.

- با توجه به مرحله ٦-١ الگوریتم، به ازای نمادa از مجموعه C مکانهایی را انتخاب می‌کنیم که نماد مطابق آنها در عبارت باقاعده a است. در این صورت p={1,3} می‌شود. زیرا در مکانهای ۱ و ۳، نماد a وجود دارد.

- با توجه به مرحله ٦-٢ الگوریتم، followpos(3) U followpos(1) را محاسبه می‌کنیم. با توجه به جدول followpos این مجموعه برابر است {1,2,3,4}. این مجموعه با مجموعه B برابر است.

- با توجه به مرحله ٦-٤ الگوریتم، dtrans[C,a]= B ،در این صورت dtrans به صورت ذیل می‌شود.

حالت	a	b
A√	B	A
B√	B	C
C√	B	

- با توجه به مرحله ٦ الگوریتم، به ازای نماد b و حالت C مراحل ذیل را انجام می‌دهیم.

- با توجه به مرحله ٦-١ الگوریتم، در این مرحله به ازای نمادb از مجموعه C مکانهایی را انتخاب می‌کنیم که نماد مطابق آنها در عبارت باقاعده b است، در این صورت p={2,5} می‌شود. زیرا در مکان ۲ و 5 نماد b قرار دارد.

- با توجه به مرحله ٦-٢ الگوریتم، followpos(2) U followpos(5) را محاسبه می‌کنیم. با توجه به جدول محاسبه followpos این مجموعه برابر است با {1,2,3,5,6}. این مجموعه، یک

مجموعه جدید است در نتیجه نام جدید D به آن منتسب کرده و آن را به جدول اضافه می‌کنیم.

حالت	a	b
A√	B	A
B√	B	C
C√	B	
D		

- با توجه به مرحله ٦-٣ الگوریتمdtrans[C,b]= D. در این صورت جدول dtrans به صورت ذیل می‌شود.

حالت	a	b
A√	B	A
B√	B	C
C√	B	D
D		

- با توجه به مرحله چهارم الگوریتم، حالت علامت نخورده بعدی را در dtrans انتخاب می‌کنیم. در این وضعیت تنها حالت علامت نخورده C است. دقت کنید که D={1,2,3,6} است.

- با توجه به مرحله پنجم الگوریتم، حالت انتخاب شده را علامت می‌زنیم.

حالت	a	b
A√	B	A
B√	B	C
C√	B	D
D√		

- با توجه به مرحله ششم به ازای نماد a و حالتD مراحل ذیل را انجام می‌دهیم.

- با توجه به مرحله ٦-١ الگوریتم، به ازای نمادa از مجموعه D مکانهایی را انتخاب می‌کنیم که نماد متناظر آنها در عبارت باقاعده a است. در این صورت p={1,3} می‌شود. زیرا در مکانهای ۱ و ۳، نماد a وجود دارد.

- با توجه به مرحله ٦-٢ الگوریتم، followpos(3) U followpos(1) را محاسبه می‌کنیم. با توجه به جدول Followpos این مجموعه برابر است {1,2,3,4}. این مجموعه با مجموعه B برابر است.

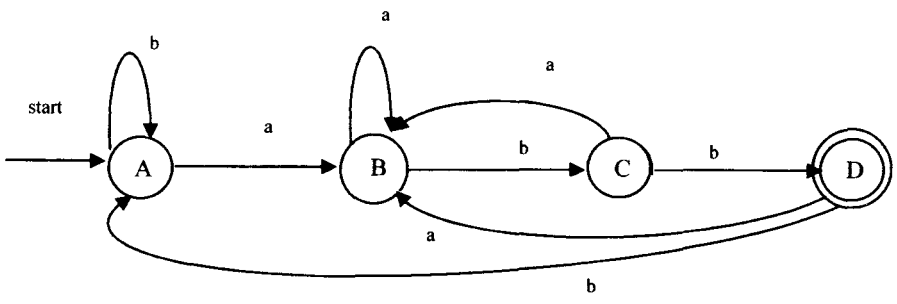
- با توجه به مرحله ٦-٤ الگوریتم، dtrans[D,a]= B ،در این صورت dtrans به صورت ذیل می‌شود.

حالت	a	b
A√	B	A
B√	B	C
C√	B	D
D√	B	

- با توجه به مرحله ٦ الگوریتم، به ازای نماد b و حالت D مراحل ذیل را انجام می‌دهیم.
- با توجه به مرحله ٦-١ الگوریتم، در این مرحله به ازای نماد b از مجموعه D مکانهایی را انتخاب می‌کنیم که نماد مطابق آنها در عبارت باقاعده b است، در این صورت p={2} می‌شود. زیرا در مکان ۲ نماد b قرار دارد.
- با توجه به مرحله ٦-۲ الگوریتم، followpos(2) را محاسبه می‌کنیم. با توجه به جدول محاسبه followpos این مجموعه برابر است با {1,2,3}. این مجموعه، یک مجموعه، جدید نیست و قبلا به عنوان A نامگذاری شده است.
- با توجه به مرحله ٦-۳ الگوریتم dtrans[D,b]= A. در این صورت جدول dtrans به صورت ذیل می‌شود.

حالت	a	B
A√	B	A
B√	B	C
C√	B	D
D√	B	A

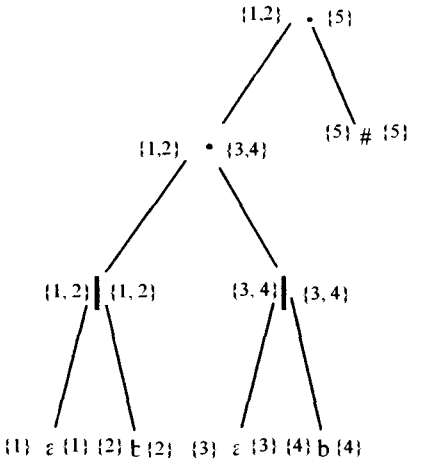
- حالت علامت نخورده ای در جدول وجود ندارد بنابراین الگوریتم پایان می‌یابد. مکان متناظر با# ، مکان 6 است، در نتیجه هر مجموعه شامل این مکان، حالت پذیرش DFA است بنابراین حالت D={1,2,3,6} ، حالت پذیرش است. DFAی حاصل به صورت ذیل است.



شکل ۲-۵۲:DFA مربوط به (a|b)*abb

مثال۲-۳٦ برای عبارت باقاعده (a|b)(a|b) مستقیما یک DFA بسازید.

ابتدا درخت دستور را برای عبارت باقاعده (a|b)(a|b) می‌سازیم. نتیجه در شکل زیر نشان داده شده است.



شکل ۲-۵۳ : درخت دستور مربوط به (a|b)(a|b)

جدول follow به صورت ذیل است.

جدول ۲-۲۳ محاسبه followpos

حالت	follow
1	{3,4}
2	{3,4}
3	{5}
4	{5}
5	-

با توجه به درخت دستور:

firstpos(root)={1,2}

مجموعه {1,2} را A می‌نامیم، این حالت، حالت شروع DFA است.

مراحل ذیل را روی A اجرا می‌کنیم.

- مکانهایی از {1,2} که نماد متناظر آنها a است مجموعه {1} است.

follow(1)={3,4}

- مجموعه {3,4} را B می‌نامیم، بنابراین:

dtrans[A,a]=B

- مکانهایی از {1,2} که نماد متناظر آنها b است مجموعه {2} است.

follow(2)={3,4}

- مجموعه {3,4} مجموعه B است که در مراحل قبلی به دست آمد. بنابراین:

dtrans[A,b]=B

در نتیجه:

	a	b
A√	B	B

مراحل ذیل را روی B اجرا می‌کنیم.

- مکانهایی از {3,4} که نماد متناظر آنها a است مجموعه {3} است.

follow(3)={5}

- مجموعه {5} را C می‌نامیم، بنابراین:

dtrans[B,a]=C

- مکانهایی از {3,4} که نماد متناظر آنها b است مجموعه {4} است.

follow(4)={5}

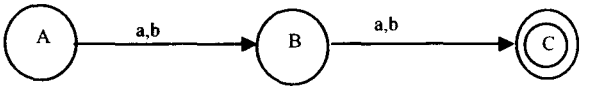
- مجموعه {5} مجموعه B است که در مراحل قبلی به دست آمد، بنابراین:

dtrans[B,b]=C

جدول ۲-۲۴ جدول تغییر حالت

	a	b
A√	B	B
B√	C	C

حالت C شامل حالت متناظر با # است در نتیجه C حالت پذیرش است.



شکل ۲-۵۴ DFA مربوط به (a|b)(a|b)

۲-۱۴ پیاده سازی DFA

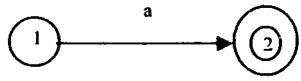
پس از تعریف ساختار لغات برنامه با عبارات باقاعده و تبدیل عبارات باقاعده به DFA قدم بعدی در ساخت تحلیلگر لغوی، تبدیل DFA به برنامه است. به این معنی که برنامه‌ای بسازیم که رشته‌ای را به عنوان ورودی دریافت کرده و مشخص نماید آیا این رشته توسط عبارت باقاعده قابل تولید است یا خیر. یا به به عبارت دیگر آیا این رشته در زبان تولیدی یک عبارت باقاعده است.

روشهای مختلفی برای پیاده سازی DFA وجود دارد که در ذیل یکی از این روشها را بررسی می‌کنیم. هر DFA شامل مجموعه‌ای از حالات و لبه‌ها است، در برنامه هر حالت را به وسیله یک case از دستور switch در زبان C (یا زبان پاسکال) پیاده سازی می‌کنیم. درون هر case نماد ورودی بعدی را تست می‌کنیم، اگر نماد مطابق برچسب لبه باشد ،حالت بعدی

مشخص می‌شود. در این روش از متغیری به نام state استفاده می‌کنیم تا حالت جاری را نشان دهد. state در ابتدا حاوی شماره حالت شروع را دارد. این روش را با ذکر چند مثال نشان می‌دهیم.

مثال۲-۳۷ r=a

ابتدا DFA مربوط به این عبارت باقاعده را رسم می‌کنیم.



شکل ۲-۵۵ DFA مربوط به عبارت باقاعده a

دراین DFA دو حالت ۱و ۲ وجود دارد، در نتیجه در دستور switch دو case در نظر می‌گیریم. از یک متغیر به نام state استفاده می‌کنیم. state در ابتدا شماره حالت شروع را خواهد داشت. state حالت جاری را نشان می‌دهد. در case شماره یک بررسی می‌کنیم اگر کاراکتر ورودی a باشد به حالت ۲ منتقل می‌شود[۱].

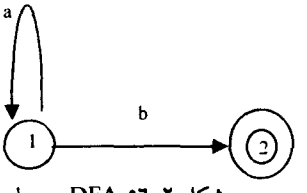
```
#inlcude<iostream.h>
#inlcude<stdlib.h>
#inlcude<stdio.h>
int main(){
int state;
char  ch;
state=1;
switch(state){
        case 1: ch= getc(stdin);
                if (ch=='a')
                        state=2;
                else {
                        cout<<"Failed";
                        exit(0);
                     }
                break;
        case 2: ch=getc(stdin);
                if(ch='\n')
                        cout<<"Accepted";
                else
                        cout<<"Failed";

                exit(0);
          }
```

۱. تابع exit(0) برای خروج از برنامه است. تابع getc(stream) یک کاراکتر را از جریان stream خوانده و برمی گرداند. getc(stdin) یک کاراکتر را از جریان stdin (جریان ورودی استاندارد یعنی صفحه کلید) می‌خواند.

```
return(0);
}
```

مثال ۲-۳۸ r=a*b

ابتدا DFA مربوط به این عبارت با قاعده را رسم می‌کنیم.



شکل ۲-۵۶ DFA مربوط به a*b

دراین DFA دو حالت ۱و ۲ وجود دارد، در نتیجه دو case در نظر می‌گیریم. با توجه به DFA در حالت یک اگر کاراکتر ورودی a باشد باید به حالت یک برگردد. برای پیاده‌سازی این قسمت، در case شماره یک بررسی می‌کنیم اگر کاراکتر ورودی a باشد state=1 قرار می‌دهیم و اگر کاراکتر ورودی b باشد به حالت ۲ منتقل می‌شود در غیر این صورت خطا خواهد بود.

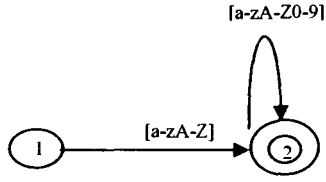
```
#inlcude<iostream.h>
#inlcude<stdlib.h>
#inlcude<stdio.h>
int main(){
int state;
char  ch;
state=1;
while(1){
switch(state){
        case 1: ch = getc(stdin);
                if (ch=='a')
                        state=1;
                else if(ch == 'b')
                        state=2;
                else {
                        cout<<"Failed";
                        exit(0);
                    }
                break;
        case 2: ch=getc(stdin);
                if(ch='\n')
                        cout<<"Accepted";
                else
                        cout<<"Failed";

                exit(0);
        }
}
return(0);
```

}

مثال۲-۳۹ برنامه بنویسید که رشته ای را دریافت کرده و مشخص کند آیا این رشته توسط عبارت باقاعده ذیل قابل تولید است یا خیر.

[a-zA-Z][a-zA-Z0-9]*

ابتدا DFA مربوط به این عبارت با قاعده را رسم می‌کنیم. به منظور خلاصه نویسی به جای نوشتن تمام برچسبهای ....a,b,c,....,A,B,C,....,0,1,2,3 از [a-zA-Z0-9] استفاده می‌کنیم. a-z شامل همه حروف کوچک از a تا z است و A-Z شامل همه حروف بزرگ از A تا Z است و 0-9 شامل همه ارقام از 0 تا 9 است.



شکل ۲-۵۷ DFA مربوط به [a-zA-Z][a-zA-Z]*

دراین DFA دو حالت ۱و ۲ وجود دارد، بنابراین دو case در نظر می‌گیریم. با توجه به DFA در حالت یک اگر کاراکتر ورودی یک حرف باشد باید به حالت دو منتقل شود. برای پیاده سازی این قسمت، در case شماره یک بررسی می‌کنیم که اگر کاراکتر ورودی یک حرف است، state=2 قرار می‌دهیم.

```
#inlcude<iostream.h>
#inlcude<stdlib.h>
#inlcude<stdio.h>
int main(){
int state;
char  ch;
state=1;
while(1){
switch(state){
        case 1: ch = getc(stdin);
                if ((ch>='a' && 'z'>=ch )|| (ch>='A' && 'Z'>=ch) )
                        state=2;
                else {
                        cout<<"Failed";
                        exit(0);
                    }
                break;
        case 2:ch = getc(stdin);
             if(ch=='\n'){
                        cout<<"Accepted";
                        exit(0);
                        }
```

```
                else if ((ch>='a' && 'z'>=ch )|| (ch>='A' && 'Z'>=ch) || (ch>='0' &&
'9'>=ch))
                state=2;
        else {
                cout<<"Failed";
                exit(0);
              }
        exit(0);
        break;
   }
return(0);
}
```

برای هر نوع لغت در زبان مبدا یک عبارت باقاعده و در نتیجه یک DFA رسم می‌شود. هر DFA را می‌توان بوسیله یک برنامه جداگانه پیاده‌سازی کرد. برای پیاده‌سازی تحلیلگر لغوی باید در یک برنامه جریان نمادهای ورودی با همه DFAها مقایسه شود.

برای این منظور نمادهای جاری با DFAی اول مقایسه می‌شود اگر این مقایسه با شکست مواجه شد با دومین DFA مقایسه می‌شود و این روند تا آخرین DFA ادامه می‌یابد، اگر رشته توسط هیچ یک از DFA ها پذیرفته نشد، دنباله ورودی غیر مجاز است و بنابراین اعلان خطا می‌شود.

مثال۲-٤۰ فرض می‌کنیم در یک زبان انواع لغات ذیل وجود داشته باشد.

۱- شناسه

۲- عدد صحیح

۳- عدد اعشاری

٤- فضای خالی

ابتدا عبارات با قاعده را برای هر یک از این لغات مشخص می‌کنیم. عبارات با قاعده به شرح ذیل است:

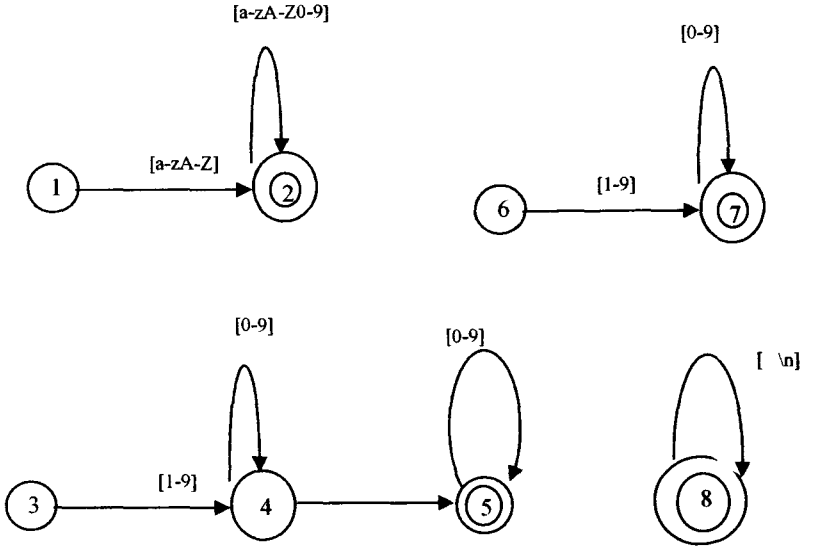
۱- شناسه:r1=[a-zA-Z][a-zA-Z0-1]*

۲- اعداد صحیح: *[0-9][1-9]

۳-اعداد اعشاری: .*[0-9].*[0-9][1-9]

٤- فضای خالی *[\n]

برای هر یک از عبارات با قاعده DFA رسم می‌کنیم.



شکل ۲-۵۸ DFA مربوط به انواع لغات

از یک تابع کمکی به نام int fail(int start) استفاده می‌کنیم، این تابع حالت شروع DFAی بعدی را مشخص می‌کند. پیاده سازی تابع fail برای مثال مورد نظر ما به صورت ذیل است.

```
int fail(int  start)
{
int  nextstart;
switch(start){
case 1:nextstart=3;  //              اعداد اعشاری
        break;
case 3:nextstart=6;   //             اعداد صحیح
        break;
case 6:nextstart=8;  //              فضای خالی
        break;
}
return  nextstart;
}
```

در این تابع اگر start=1 باشد، با توجه به DFAی دوم start=3 است و اگر start=3 باشد حالت شروع DFAی بعدی یعنی ۷ را باز می‌گرداند. اگر start=8 باشد در این صورت تمام DFA ها تست شده است و جواب اخذ نشده است.

برنامه ذیل رشته های درون فایل source.txt را خوانده و با DFA ها مقایسه می‌کند و متناسب با هر رشته (به جز فضای خالی) نشانه مربوطه را چاپ می‌کند. متغیر start مشخص می‌کند رشته با کدامیک از DFA ها مقایسه می‌شود، به عنوان مثال اگر start=1 باشد، رشته ورودی با DFA شناسه مقایسه می‌شود اگر این مقایسه با شکست مواجه گردد تابع fail فراخوانی شده و DFAی بعدی را به وسیله start=3 مشخص می‌کند و این روند تا DFAی آخر ادامه می‌یابد. پس از کشف هر رشته و چاپ نشانه مربوطه، start=1 می‌گردد تا مقایسه دوباره از اولین DFA شروع گردد.

```
#include<iostream.h>
#include<stdlib.h>
#include<stdio.h>
int fail(int start){
int nextstart;
switch(start){
case 1:nextstart=3;
     break;
case 3:nextstart=6;
     break;
case 6:nextstart=8;
     break;
     }
return (nextstart);
}
int main{
int  state,start,loc;
char ch;
FILE *fp;
fp=fopen("source.txt","r");
state=1;
start=1;
while(1)
switch(state){
```

پیاده سازی DFA اول //

```
case 1: loc=ftell(fp);
          ch=getc(fp);
          if((ch>='a' && ch<='z') || (ch>='A' && ch<='Z'))
            state=2;
          else{
                start=fail(start);
              state=start;
              fseek(fp,loc,SEEK_SET);
             }
          break;
case 2:ch=getc(fp);
     if((ch>='a' && ch<='z') || (ch>='A' && ch<='Z') || (ch>='0' && ch<='9'))
                    state=2;
     else if (ch==' '|| ch=='\n'|| ch==EOF){
              cout<<"ID;"
```

```
            state=8;
}
   else{
            start=fail(start);
            state=start;
            fseek(fp,loc,SEEK_SET);
}
    break;
```

پیاده سازی DFA دوم //

```
case 3: loc=ftell(fp);
          ch=getc(fp);
          if(ch>='1' && ch<='9')
            state=4;
          else {
                    start=fail(start);
                    state=start;
                    fseek(fp,loc,SEEK_SET);
            }
          break;
case 4: ch=getc(fp);
          if(ch=='.')
             state=5;
          else if(ch>='0' && ch<='9')
            state=4;
          else { start=fail(start);
               state=start;
               fseek(fp,loc,SEEK_SET);
               }
          break;
case 5: ch=getc(fp);
          if(ch>='0' && ch<='9')
            state=5;
          else if(ch==' '|| ch=='\n'|| ch==EOF)
          {
                    cout<<"REAL";
                    state=8;
            }
          else{
               start=fail(start);
               state=start;
               fseek(fp,loc,SEEK_SET);
          }
          break;
```

پیاده سازی DFA سوم //

```
case 6: loc=ftell(fp);
          ch=getc(fp);
          if(ch>='1' && ch<='9')
            state=7;
          else {
                    start=fail(start);
                    state=start;
                    fseek(fp,loc,SEEK_SET);
               }
          break;
```

```
case 7: ch=getc(fp);
        if(ch>='0' && ch<='9')
          state=7;
        else if(ch==' ' || ch=='\n'|| ch==EOF)
           {
                cout<<"NUM;"
                state=8;
        }
        else { start=fail(start);
             cout<<"error in input stream;"
             exit(0);
        }
        break;
```

پیاده سازی DFA چهارم //

```
case 8: ch=getc(fp);
        if(ch==' '|| ch=='\n')
                  state=8;
        else if(ch==EOF )
                  exit(0);
        else {
              ungetc(ch,fp);
                 start=1;
                 state=1;
        }
        break;
}
return 0;
}
```

این تحلیلگر لغوی به صورت یک برنامه نوشته شده است. به سادگی می‌توان این برنامه را به تابعی تبدیل کرد که با هر فراخوانی یک لغت را تشخیص دهد.

ممکن است بخشی از یک دنباله از کاراکترها با یک عبارت باقاعده و بخش دیگری عبارت باقاعده دیگری منطبق شود. به عنوان مثال دنباله 123.65 با دو عبارت با قاعده ذیل را در نظر می‌گیریم.

```
[1-9][0-9]*
[1-9][0-9]*. [0-9][1-9]*
```

دنباله 123 از 123.65 با *[0-9][1-9] و دنباله 123.65 با *[1-9][0-9] .*[0-9][1-9] منطبق است. تحلیلگر لغوی باید مکانیزمی در اختیار داشته باشد که بتواند بین این دو حالت یکی را انتخاب کند. قانونی که بکار می‌رود، پذیرش طولانی ترین دنباله است، در نتیجه تحلیلگر لغوی دنباله 123.65 را به جای 123 با توجه به *[1-9][0-9] .*[0-9][1-9] می‌پذیرد، زیرا طولانی تر از رشته 123 است.

همچنین ممکن یک دنباله از کاراکترها (نه بخشی از یک دنباله) با دو یا چند عبارت باقاعده منطبق شود به عنوان مثال رشته if می‌تواند توسط دو عبارت باقاعده ذیل تولید شود.

```
if
[a-zA-Z][a-zA-Z0-9]*
```

در این وضعیت نیاز به روشی است که تحلیلگر لغوی بتواند بر اساس آن یکی از دو عبارت را تعیین کند. برای حل این مشکل از اولویت استفاده می‌کنیم. یعنی برای عبارات با قاعده اولویت در نظر می‌گیریم. برای تعیین اولویت از ترتیب مکانی استفاده می‌شود. عبارت با قاعده ای که اولویت بیشتری دارد، ابتدا مقایسه می‌شود. در مثال بالا دنباله if با عبارت باقاعده if منطبق می‌شود. ولی اگر عبارات باقاعده به ترتیب ذیل باشند.

```
[a-zA-Z][a-zA-Z0-9]*
if
```

تحلیلگر لغوی [a-zA-Z][a-zA-Z0-9]* را انتخاب می‌کند.

اولویت در پیاده سازی تحلیلگر لغوی باید لحاظ شود. تابع ()fail، DFAی بعدی را مشخص می‌کرد با تغییر این تابع می‌توان ترتیب مقایسه DFAها را تغییر داد. به عنوان مثال با تغییر تابع fail به صورت ذیل ابتدا اعداد صحیح و سپس اعداد اعشاری تست خواهد شد(در حالت قبلی ابتدا اعداد اعشاری و سپس اعداد صحیح تست شد).

```
int fail(int start){
int nextstart;
switch(start){
case 1:nextstart=6; //          اعداد صحیح
      break;
case 3:nextstart=8;
      break;
case 6:nextstart=3;  //          اعداد اعشاری
      break;
    }
return (nextstart);
}
```

۲-۱۵ کلمات کلیدی

برای تشخیص کلمات کلیدی دو روش وجود دارد:

۱- با هر کلمه کلیدی به طور مستقل برخورد می‌کنیم. برای هر کلمه کلیدی یک عبارت باقاعده در نظر می‌گیریم و سپس برای آن DFA را به دست آورده و سپس آن را با دیگر DFAها پیاده‌سازی می‌نماییم. در این روش باید ابتدا نمادهای ورودی با عبارات باقاعده مربوط به کلمات کلیدی و سپس با عبارات با قاعده مربوط به دیگر انواع لغات مقایسه شوند.

۲- از آنجاییکه کلمات کلیدی از لحاظ ساختار زیر مجموعه شناسه‌ها هستند می‌توان آنها را مانند شناسه‌ها تشخیص داد، با این تفاوت که کلمات کلیدی در جدول نماد به عنوان مقدار اولیه ثبت می‌شوند و نشانه کلمات کلیدی نیز در جدول نماد ثبت می‌شود تحلیلگر لغوی هر بار که شناسه‌ای را تشخیص داد ابتدا جدول نماد را جستجو می‌کند. اگر شناسه کلمه کلیدی باشد تحلیلگر لغوی با جستجو در جدول نماد کلمه کلیدی را یافته و نشانه ثبت شده در جدول نماد را به عنوان نشانه برمی‌گرداند.

جدول۲-۲۵ نمایش کلمات کلیدی در جدول نماد

لغت	نشانه	صفات
program	PROG_TK	
var	VAR_TK	
integer	INTEGER_TK	
real	REAL_TK	
function	FUNCTION_TK	

۲-۱۶ تولید خودکار تحلیلگر لغوی

برای ساخت تحلیلگر لغوی روشهای مختلفی وجود دارد که عبارتند از:

الف- استفاده از زبانهای برنامه‌سازی: در این روش با استفاده از یکی از زبانهای برنامه سازی (مانندC یا پاسکال) تحلیلگر لغوی ساخته می‌شود. این روش دارای معایب و مزایایی است که عبارتند از:

۱- صرف زمان زیاد برای ساخت تحلیلگر لغوی

۲- کاهش قابلیت استفاده مجدد

۳- افزایش امکانات

٤- افزایش قابلیت انعطاف پذیری

ب- استفاده از ابزارها: با توجه به اینکه اصول ساخت تحلیلگر لغوی برای کامپایلرها یکسان است ابزارهایی به منظور تولید تحلیلگر لغوی ایجاد شده است. در این روش از ابزارهایی که به منظور تولید تحلیلگر لغوی ایجاد شده اند استفاده می‌گردد. استفاده از ابزارها دارای مزایا و معایبی است که عبارتند از:

۱- افزایش سرعت ایجاد تغییرات

۲- کاهش زمان ساخت تحلیلگر لغوی

۳- افزایش محدودیتها

در بخش قبل روش پیاده‌سازی بوسیله زبان برنامه‌سازی (مانند زبان C) بررسی شد. در این بخش، تولید تحلیلگر لغوی به وسیله ابزارها را مورد بررسی قرار می‌دهیم.

۲-۱۷ پیاده سازی تحلیلگر لغوی به وسیله تولیدکننده تحلیلگر لغوی[1]

اصول و روشهای ساخت تحلیلگر لغوی برای زبانهای مختلف یکسان است، در نتیجه ابزارهایی ایجاد شده است که با اخذ مشخصات لغوی زبان مبدا، می‌توانند تحلیلگر لغوی آن را تولید می‌کنند. این گونه نرم افزارها را تولید کننده تحلیلگر لغوی می‌نامیم. یکی از مهمترین این ابزارها نرم افزار Flex[2] است. به دلیل اهمیت ابزارها، در این قسمت تولیدکننده تحلیلگر لغوی Flex را مورد بررسی قرار می‌دهیم. Flex را می‌توانید از طریق اینترنت جستجو کرده و بدست آورید[3].

۲-۱۸ استفاده از Flex

Flex شامل یک زبان برنامه‌نویسی خاص خود است. FLex یک کامپایلر است که برنامه به زبان Flex را به زبان C یا پاسکال تبدیل می‌کند. برنامه نویس ساختار لغات مورد نظر خود را به وسیله عبارات با قاعده در یک فایل متنی (در یک ویرایشگر متن مانند Notepad) به زبان Flex بیان می‌کند، پسوند این فایل باید L. باشد، Flex با خواندن این برنامه تحلیلگر لغوی مورد نظر برنامه نویس را به زبان C تولید می‌کند. با کامپایل برنامهC، تحلیلگر لغوی تولید می‌گردد[4]. شکل ۲-۵۹ ذیل مراحل استفاده از Flex را جهت تولید تحلیلگر لغوی را نشان می‌دهد.

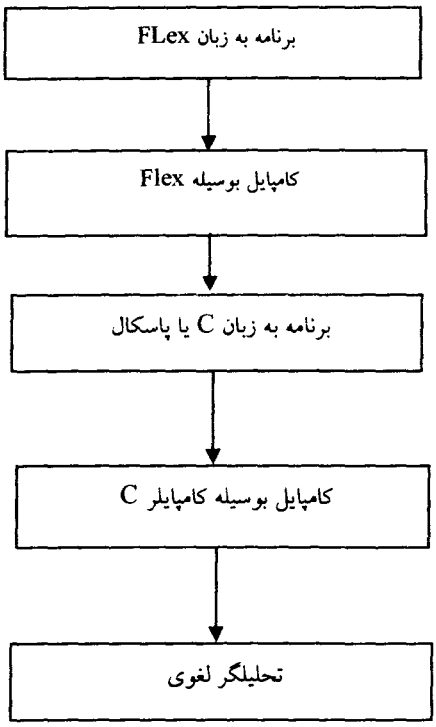
---

1. Lexical Analyzer Generator

۲. Flex (Fast lex) نسخه‌ای از نرم افزار lex است. برای مباحث این بخش از نرم افزار lex نیز می‌توانید استفاده کنید.

۳. حجم نرم افزار Flex کم است به همین جهت می توانید آن را از اینترنت Download کنید. البته معمولا flex را در کنار نرم افزار Bison خواهید یافت. همچنانکه نرم افزار lex را در کنار نرم افزار yacc می یابید. lex و flex تولید کننده تحلیلگر لغوی و Bison و yacc تولیدکننده تحلیلگر نحوی هستند.

٤. البته نسخه‌هایی از lex و flex وجود دارند که برنامه را به زبان پاسکال تولید می کنند.



شکل ۲-۵۹ استفاده از flex

برای تولید تحلیلگر لغوی از دستور ذیل در خط فرمان استفاده می‌گردد.

flex نام فایل مبدا نام فایل مقصد

نام فایل مبدا: نام فایل حاوی مشخصات لغوی زبان مبدا

نام فایل مقصد: نام فایل حاوی برنامه تحلیلگر لغوی به زبان C

Flex برنامه مبدا را خوانده و برنامه تحلیلگر لغوی را به زبان C تولید کرده و در فایل مقصد ذخیره می‌کند. اگر نام فایل مقصد مشخص نگردد، Flex برنامه تولیدی را در فایل lexyy.c ذخیره می‌کند. برای کامپایل برنامه C از کامپایلرهای زبان C مانند Tourbo C و یا Borland C استفاده می‌شود.

مثال۲-٤۱ فرض کنید Pascal.l فایل حاوی مشخصات لغوی زبان پاسکال است. برای تولید تحلیلگر لغوی از دستور ذیل در خط فرمان استفاده می‌کنیم.

```
c:\> flex   pascal.l    scanner.c
```

در این مثال فرض می‌کنیم که flex در مسیر c:\ قرار دارد. نتیجه اجرای این دستور برنامه scanner.c است که حاوی برنامه تحلیلگر لغوی به زبان C است. پس از کامپایل scanner.c با استفاده از کامپایلرهای C ، تحلیلگر لغوی scanner.exe به دست می‌آید. برای استفاده از scanner.exe دو روش وجود دارد:

۱- رشته هایی که باید توسط تحلیلگر لغوی (scanner.exe) مورد ارزیابی قرار گیرد را می‌توان داخل یک فایل قرار داده و سپس در اختیار تحلیلگر لغوی قرار می‌دهیم. به عنوان مثال فرض کنید برنامه به زبان مبدا در فایل p1 قرار دارد در این صورت از دستور ذیل استفاده می‌کنیم.

```
c:\> scanner.exe< p1
```

scanner.exe محتوای فایل P1 را خوانده و مورد پردازش قرار می‌دهد و خروجی را در مانیتور نشان می‌دهد. می‌توان با استفاده از دستورات DOS امکانی فراهم کرد که خروجی نیز به یک فایل منتقل شود.

```
c:\> scanner.exe  <p1> result
```

در این صورت خروجی برنامه در داخل فایل result قرار می‌گیرد.

۲- رشته ها را می‌توان در خط فرمان وارد کرد.

```
c:\> scanner.exe        رشته های مورد نظر
```

۲-۱۸-۱ نحوه بیان عبارات با قاعده

ساختار لغات برنامه مبدا را به وسیله عبارات با قاعده می‌توان بیان کرد. Flex امکانی فراهم می‌کند که عبارات با قاعده را دریافت کرده و تحلیلگر لغوی مطابق آن را می‌سازد. Flex اطلاعات را به صورت جریانی از کاراکترهای اسکی دریافت کرده و با عبارات باقاعده مقایسه می‌کند. بخش مهم ساخت تحلیلگر لغوی بیان ساختار لغات به زبان Flex است. چندین روش برای بیان عبارات باقاعده وجود دارد. ساده‌ترین روش، استفاده از رشته‌ها است. اگر یک دنباله مشخص از کاراکترها (مانند کلمات کلیدی if، while) مد نظر باشد می‌توان رشته را داخل علامت " قرار داد. به مثالهای ذیل توجه فرمایید:

"if"	عبارت با قاعده if
"while"	عبارت با قاعده while
"goto"	عبارت با قاعده goto

در Flex می‌توان از کاراکترهای ویژه ذیل نیز استفاده نمود:

\n: برای نشان دادن سر خط

\b: برای نشان دادن فضای خالی

\tab: برای نشان دادن tab

در جدول ۲-۲۶ نحوه بیان عبارات باقاعده به زبان flex ارائه شده است.

یک برنامه به زبان flex شامل سه قسمت اصلی است. که هر قسمت توسط علامت %% از قسمت دیگر جدا شده است.

تعاریف[1]
%%
ترجمه ها[2]
%%
توابع[3]

۲-۱۸-۱-۱ تعاریف

این بخش در زبان Flex اختیاری است یعنی می‌توان برنامه به زبان FLex نوشت که شامل تعاریف نباشد. در این قسمت نامهایی که در قسمتهای دیگر استفاده می‌شوند تعریف می‌شوند. به عنوان مثال می‌توان عبارات باقاعده را تعریف کرده و به آنها نامی اختصاص دهیم و در قسمتهای دیگر از آن استفاده کنیم. شکل کلی هر عبارت در این قسمت به صورت ذیل است.

نام عبارت باقاعده

مثال ۲-۴۲

```
lower   [a-z]
upper   [A-Z]
letter  lower|upper
```

در این مثال به عبارت با قاعده [a-z] نام lower و به عبارت با قاعده [A-Z] نام upper را تخصیص می‌دهد. سپس با استفاده از این دو نام به عبارت با قاعده lower|upper نام letter تخصیص داده شده است.

مثال ۲-۴۳

```
digit   [0-9]
lower   [a-z]
upper   [A-Z]
letter  lower|upper
var     {letter}|({letter}|{digit})*
```

1. Definitons 2. Translations 3. Functions

جدول ۲-۲۶ نحوه بیان عبارات باقاعده به زبان flex

عبارت باقاعده	رشته‌های منطبق بر عبارت باقاعده	مثال
	نشان دهنده هر کاراکتر به جز کاراکتر سرخط[1] است.	
x	رشته های یک کاراکتری که با کاراکتر x منطبق می‌شود.	a این عبارت با قاعده فقط کاراکتر a را می‌پذیرد.
[مجموعه کاراکترها]	رشته های یک کاراکتری که با هر یک از کاراکترهای مشخص شده منطبق شوند.	[xwa] این عبارت با قاعده فقط رشته های یک کاراکتری x,w و یا a را می‌پذیرد.
[کاراکتر مقصد- کاراکترمبدا]	رشته‌های یک کاراکتری که با هر یک از کاراکترهای بین کاراکتر مبدا و کاراکتر مقصد مشخص شده منطبق شوند.	[a-f] این عبارت با قاعده فقط رشته های یک کاراکتری که منطبق بر a,b,c,d,e,f منطبق باشند.
r*	صفر الی چند تکرار از r (r یک عبارت باقاعده است)	a* رشته هایی که شامل صفر الی چند a باشند.
r+	یک الی چند تکرار از r (r یک عبارت باقاعده است)	a+ رشته هایی که شامل یک الی چند a باشند.
r?	صفر الی یک تکرار از r (r یک عبارت باقاعده است)	a? رشته هایی که شامل صفر یا یک کاراکتر a باشند.
r{m,n}	حداقل m و حداکثر n تکرار از r (بین m تا n تکرار از r)	x{3,7} رشته هایی که بین ۳ تا ۵ کاراکتر x داشته باشند.
r1|r2	رشته هایی که با r1 یا r2 منطبق شوند.	a+|b+ رشته هایی که یک یا بیشتر a و یا رشته هایی که یک یا بیشتر b داشته باشند.
r1r2	رشته هایی که قسمت اول آن با r1 و قسمت دوم آنها یا r2 منطبق شوند	a+b+ رشته های که شامل یک یا بیشتر a به دنبال آن یک یا بیشتر b داشته باشند.
r1/r2	رشته هایی که با r1 منطبق شوند به شرطی که به دنبال آن رشته ای باشد که با r2 منطبق شوند.	
[کاراکترهای مورد نظر^]	هر کاراکتری به جز کاراکترهای مشخص شده	[^ab] همه رشته های یک کاراکتری به جز a و b

علاوه بر نامگذاری عبارات باقاعده اعلانهای مورد نیاز (مانند تعریف متغیرها) توابع در بخشهای دیگر در این قسمت معرفی می‌شوند. این بخش درون }% و {% قرار می‌گیرد. به مثال ذیل دقت کند.

```
%{
#include<stdio.h>
int nchar,nline;
```

2. New Line

```
%}
```

۲-۱۸-۱-۲ ترجمه

از بخشهای ضروری برنامه flex بخش ترجمه است. یعنی هر برنامه flex باید شامل قسمت ترجمه باشد. این قسمت مشخص می‌کند هنگام کشف یک لغت مطابق یکی از عبارات با قاعده چه عملی باید انجام شود. این بخش شامل عباراتی به صورت ذیل است.

الگوی نشانه {عملیات}

الگوی نشانه: عبارت باقاعده و یا یکی از اسامی تعریف شده در بخش تعاریف است.

عملیات: دستورالعملهایی به زبان C را نشان می‌دهد که هنگام یافتن دنباله‌ای از کاراکترها مطابق الگوی نشانه باید اجرا شوند.

مثال ۲-٤٤

```
digit    [0-9]
lower    [a-z]
upper    [A-Z]
letter   lower|upper
var      {letter}|({letter}|{digit})*
ws       [ \n\t]+
%%
ws       {}
"if"     {printf("I found 'IF' keyword");}
"else"   {printf("I found 'ELSE' keyword");}
var      {printf("I found  variable ");}
%%
```

هرگاه تحلیلگر لغوی دنباله if را کشف کند عبارت I found 'IF' keyword چاپ می‌شود[1] و هرگاه دنباله else کشف شود I found 'ELSE' keyword چاپ خواهد شد و به همین ترتیب وقتی به یک متغیر کشف شود عبارت I found variable چاپ خواهد شد و برای فضای خالی عملیاتی انجام نمی‌دهد.

مثال ۲-٤٥ اگر متن ذیل به عنوان ورودی به این تحلیلگر داده شود.

```
if  temp  else  if  id 34
```

خروجی به صورت ذیل خواهد بود.

```
I found 'if' keyword
I found variable
I found 'ELSE' keyword
I found 'if' keyword
I found variable
```

همانطور که ملاحظه می‌شود به ازای هر بار کشف یک لغت عمل مشخص شده انجام می‌شود.

---

۱. printf تابعی است که برای چاپ رشته ها در مانیتور می توان از آن استفاده کرد.

۲-۱۸-۱-۳ توابع

این قسمت اختیاری می‌باشد. در این قسمت توابع مورد نیاز برنامه ایجاد می‌شوند.

flex تحلیلگر لغوی را به یک تابع به نام yylex() تبدیل می‌کند. برای اجرای تحلیلگر لغوی yylex(); فراخوانی می‌گردد. بنابراین در تابع main() این تابع فراخوانی می‌شود.

مثال۲-٤٦ برنامه ای به زبان flex بنویسید که تعداد کاراکترها و خطوط ورودی را شمارش کرده و نتیجه را چاپ کند.

```
%option  noyywrap
%{
int nchar,nline;
%}
%%
[\n]     {nline++;}
.          {nchar++;}
}
%%
int main(void){
yylex();
printf("%d     %d",nchar,nline+1);
return (0);
}
```

مثال۲-٤٧ برنامه ذیل اغلب لغات برنامه به زبان پاسکال را شناسایی کرده و بر حسب مورد پیغام مناسب را چاپ می‌کند.

```
%option noyywrap
NQUOTE   [^']
%%
[a-zA-Z]([a-zA-Z0-9])*          printf("ID");
":="                            printf("ASSIGNMENT ");
'({NQUOTE}|'')+'                printf("CHARACTER_STRING ");
":"                             printf("COLON ");
","                             printf("COMMA ");
[0-9]+                          printf("DIGSEQ ");
"."                             printf("DOT ");
".."                            printf("DOTDOT ");
"="                             printf("EQUAL ");
">="                            printf("GE ");
">"                             printf("GT ");
"["                             printf("LBRAC ");
"<="                            printf("LE ");
"("                             printf("LPAREN ");
"<"                             printf("LT ");
"-"                             printf("MINUS ");
"<>"                            printf("NOTEQUAL ");
"+"                             printf("PLUS ");
"]"                             printf("RBRAC ");
```

```
[0-9]+"."[0-9]+          printf("REALNUMBER ");
")"                      printf("RPAREN ");
";"                      printf("SEMICOLON ");
"/"                      printf("SLASH ");
"*"                      printf("STAR ");
"**"                     printf("STARSTAR ");
"^"                      printf("UPARROW ");
[ \n\t\f]+               ;
%%
int main(){
yylex();
return 0;
}
```

پس از نوشتن این برنامه در notepad و ذخیره این فایل به نام دلخواه مانند pascallex.l این برنامه را توسط دستور ذیل به زبان C تبدیل می‌کنیم.

```
C:\flex pascallex.l
```

از آنجاییکه فایل مقصد مشخص نشده است. در نتیجه برنامه تولیدی در فایل lexyy.c است. پس از کامپایل lexyy.c برنامه lexyy.exe به دست می‌آید، که تحلیلگر لغوی مورد نظر است. در این برنامه کلمات کلیدی توسط عبارت باقاعده [a-zA-Z]([a-zA-Z0-9])* پذیرفته می‌شود. در نتیجه برای کلمات کلیدی نیز ID چاپ می‌شود. برای تست این برنامه، فایل p1.pas را با محتوای ذیل در نظر می‌گیریم.

```
program test;
var i:integer;
begin
i:=2;
end.
```

برای استفاده از lexyy.exe از دستور ذیل استفاده می‌کنیم.

```
c:\lexyy.exe<p1.pas>t1
```

این دستور باعث می‌شود که lexyy.exe ورودی را به جای صفحه کلید از فایل p1.pas بخواند، و خروجی را به جای مانیتور در فایل t1 قرار دهد. محتوای t1 در ذیل نشان داده شده است.

```
ID ID SEMICOLON ID ID COLON ID SEMICOLON IDID ASSIGNMENT  DIGSEQ
SEMICOLON ID DOT
```

در برخی موارد ممکن است ناسازگاریهایی بروز کند به عنوان مثال ممکن است بخشهای مختلف یک دنباله توسط عبارات باقاعده مختلف پذیرفته شود به عنوان مثال دنباله <> و دوعبارت باقاعده ذیل را در نظر می‌گیریم.

```
"<"                      printf("LT ");
"<>"                     printf("NOTEQUAL ");
```

> با عبارت باقاعده اول و <> با عبارت باقاعده دوم منطبق می‌گردد. در این گونه موارد تحلیلگر لغوی طولانی ترین رشته یعنی <> را با توجه به عبارت باقاعده دوم می‌پذیرد و NOTEQUALچاپ می‌شود.

۲-۱۸-۲ کلمات کلیدی

به منظور تشخیص کلمات کلیدی دو روش وجود دارد که در ذیل به شرح هر یک می‌پردازیم.

روش اول: هر کلمه کلیدی را به عنوان یک نوع لغت در نظر گرفته و عبارت باقاعده مناسب را در لیست عبارات باقاعده درج می‌کنیم. برنامه ذیل با این رویکرد ایجاد شده است.

```
%option noyywrap
NQUOTE   [^']
%%
"program"                   printf("PROGRAM ");
"var"                       printf("VAR ");
"begin"                     printf("BEGIN ");
"end"                       printf("END ");
[a-zA-Z]([a-zA-Z0-9])*      printf("ID ");
":="                        printf("ASSIGNMENT  ");
'({NQUOTE}|'')+'            printf("CHARACTER_STRING  ");
":"                         printf("COLON ");
","                         printf("COMMA ");
[0-9]+                      printf("DIGSEQ ");
"."                         printf("DOT ");
".."                        printf("DOTDOT ");
"="                         printf("EQUAL ");
">="                        printf("GE ");
">"                         printf("GT ");
"["                         printf("LBRAC ");
"<="                        printf("LE ");
"("                         printf("LPAREN ");
"<"                         printf("LT ");
"-"                         printf("MINUS ");
"<>"                        printf("NOTEQUAL ");
"+"                         printf("PLUS ");
"]"                         printf("RBRAC ");
[0-9]+"."[0-9]+             printf("REALNUMBER ");
")"                         printf("RPAREN ");
";"                         printf("SEMICOLON ");
"/"                         printf("SLASH ");
"*"                         printf("STAR ");
"**"                        printf("STARSTAR ");
"^"                         printf("UPARROW ");
[ \n\t\f]  +                ;
%%
int main(){
yylex();
return 0;
```

}

در این برنامه برای کلمات کلیدی program ، var ، begin و end عبارت باقاعده در نظر گرفته شده است. کلمات کلیدی دیگر را نیز می‌توان به این روش به برنامه اضافه کرد.

مکان تعریف عبارات با قاعده در برنامه مهم است. تحلیلگر لغوی تولیدی flex ، مقایسه دنباله ورودی را با عبارات باقاعده به ترتیب از بالا به پایین انجام می‌دهد بنابراین عبارات با قاعده کلمات کلیدی باید قبل از عبارت باقاعده شناسه باشند، در غیر این صورت تمام کلمات کلیدی، شناسه در نظر گرفته می‌شوند. به عنوان مثال ترتیب ذیل تمام کلمات کلیدی را شناسه در نظر می‌گیرد.

```
[a-zA-Z]([a-zA-Z0-9])*        printf("ID ");
"program"                     printf("PROGRAM ");
"var"                         printf("VAR ");
"begin"                       printf("BEGIN ");
"end"                         printf("END ");
```

کلمات کلیدی (مانند begin) ابتدا با عبارت باقاعده [a-zA-Z]([a-zA-Z0-9])* مقایسه می‌شوند زیرا این عبارت باقاعده در ابتدا قرار گرفته است، بنابراین begin توسط عبارت باقاعده ID پذیرفته می‌شود و با عبارت باقاعده "begin" مقایسه ای انجام نمی‌شود. با فایل ورودی p1.pas خروجی ذیل ایجاد می‌شود.

```
ID ID SEMICOLON ID ID COLON ID SEMICOLON ID ID ASSIGNMENT DIGSEQ
SEMICOLON ID DOT
```

روش دوم: کلمات کلیدی را در یک جدول (یا جدول نماد) به عنوان مقدار اولیه وارد می‌کنیم، هرگاه یک شناسه کشف می‌شود، این شناسه در جدول جستجو می‌شود اگر این شناسه با کلمات کلیدی جدول یکسان باشد، شناسه، یک کلمه کلیدی است. به عنوان مثال جدول را می‌توان به صورت ذیل تعریف کرد.

```
char keyword[40][20]= {"AND", "ARRAY" ,"BEGIN" ,"CASE" ,"CONST"
,"DIV","DO","DOWNTO","ELSE","END","EXTERNAL","EXTERN","FILE", "FOR"
,"FORWARD","FUNCTION","GOTO","IF","IN","LABEL","MOD","NIL","NOT","OF"
,"OR","OTHERWISE", "PROCEDURE",
"PROGRAM","RECORD","REPEAT","THEN","TO","TYPE","UNTIL","VAR","WHIL
E","WITH"};
```

برنامه ذیل با استفاده از روش دوم ایجاد شده است.

```
%{
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<stdlib.h>
char c;
void toupper(char k[])
{ int i;
for(i=0;i<=strlen(k);++i)
```

```
if(k[i]<='z' && k[i]>='a')
k[i]-=32;
}
int is_keyword(char id[]) {
char   keyword[40][20]={"AND",  "ARRAY"  ,"BEGIN","CASE"  ,"CONST"  ,"DIV"
,"DO","DOWNTO","ELSE","END","EXTERNAL","EXTERN","FILE",      "FOR"
,"FORWARD","FUNCTION","GOTO","IF","IN","LABEL","MOD","NIL","NOT","OF"
,"OR","OTHERWISE",                                                "PROCEDURE",
"PROGRAM","RECORD","REPEAT","THEN","TO","TYPE","UNTIL","VAR","WHIL
E","WITH"};
int i;
for(i=0;i<40;i++)
 if(strcmp(id,keyword[i])==0)
    return i;
 return -1;
}
%}
%option noyywrap
NQUOTE   [^']
%%
[a-zA-Z]([a-zA-Z0-9])*    {toupper(yytext);
                  if (is_keyword(yytext)!=-1)
                   printf("keword=%s",yytext);
                  else
                   printf("ID ");
                   }
":="                        printf ("ASSIGNMENT ");
'({NQUOTE}|'')+'            printf ("CHARACTER_STRING ");
":"                         printf ("COLON ");
","                         printf ("COMMA ");
[0-9]+                      printf ("DIGSEQ ");
"."                         printf ("DOT ");
".."                        printf ("DOTDOT ");
"="                         printf ("EQUAL ");
">="                        printf ("GE ");
">"                         printf ("GT ");
"["                         printf ("LBRAC ");
"<="                        printf ("LE ");
"("                         printf ("LPAREN ");
"<"                         printf ("LT ");
"-"                         printf ("MINUS ");
"<>"                        printf ("NOTEQUAL ");
"+"                         printf ("PLUS ");
"]"                         printf ("RBRAC ");
[0-9]+"."[0-9]+             printf ("REALNUMBER ");
")"                         printf ("RPAREN ");
";"                         printf ("SEMICOLON ");
"/"                         printf ("SLASH ");
"*"                         printf ("STAR ");
"**"                        printf ("STARSTAR ");
"^"                         printf ("UPARROW ");
"(*"                       do{ c=input();
                              if(c=='*'){
                         c=input();
                                 if (c==')')
```

```
                    break;
                else unput(c);
                            }
                        }while(1);
"{"                     while((c=input())!='}');
[ \n\t]                 ;
.                       printf("Error: ILLEGAL   ");
%%
int main(){
yylex();
return 0;
}
```

flex شامل چندین متغیر عمومی است که می‌توان در برنامه از آنها استفاده کرد. متغیر yytext یکی از متغیرهای flex است که حاوی لغتی است که توسط عبارات باقاعده پذیرفته شده است. متغیر دیگر yylen است که حاوی طول رشته پذیرفته شده است. به عنوان مثال به عبارت ذیل دقت کنید.

```
[a-zA-Z][a-zA-Z0-9]*
```

اگر دنباله TEMP توسط این عبارت باقاعده پذیرفته شود، yytext حاوی temp و yylen حاوی عدد 4 (طول رشته temp) است.

تابع input یک کاراکتر از ورودی می‌خواند وunput یک کاراکتر را به بافر ورودی بازمی‌گرداند. در این برنامه، کلمات کلیدی با حروف بزرگ در جدول نگهداری می‌شوند. با شناسایی هر شناسه ابتدا با استفاده از تابع toupper ، حروف شناسه را به حروف بزرگ تبدیل کرده و سپس با محتوای جدول مقایسه می‌شود. استفاده از toupper باعث می‌شود تحلیلگر لغوی حساس به حالت نباشد (مانند زبان پاسکال که حساس به حالت نیست). اگر از تابع toupper استفاده نگردد تحلیلگر لغوی حساس به حالت می‌شود(چنانچه زبان c اینچنین است). اگر دنباله ای از کاراکترهای ورودی توسط هیچیک از عبارات باقاعده پذیرفته نشود، خطای لغوی رخ داده است. به منظور کشف این نوع لغات و انجام عملیات لازم برای آن از علامت "." به عنوان آخرین عبارت استفاده می‌شود. به عبارت ذیل دقت کنید.

```
printf("Error: ILLEGAL   ");
```

عبارت "." به معنی هر کاراکتر به جز سرخط است. وجود این عبارت باعث کشف هر کاراکتر غیر مجاز می‌شود. در این تعریف هر کاراکتری که توسط عبارات باقاعده قبل از نقطه پذیرفته نشود توسط این عبارت پذیرفته خواهد شد در نتیجه لازم است این عبارت، آخرین دستور باشد.

همانطور که ملاحظه گردید تحلیلگرهای لغوی ساخته شده با کشف هر نوع لغت، پیغام مناسبی چاپ می‌کند. اما در یک کامپایلر واقعی تحلیلگر لغوی به جای چاپ یک پیغام باید یک نشانه مشخص را برای تحلیلگر نحوی ارسال کند. به این منظور به جای چاپ یک پیغام

می‌توان از دستور return استفاده کرد. در این شرائط تحلیلگر لغوی با کشف هر لغت نشانه مناسب را باز می‌گرداند. قطعه برنامه ذیل به این منظور ارائه شده است. در فصل آینده نحوه استفاده تحلیلگر نحوی از این نشانه‌ها را توضیح می‌دهیم.

```
%%
[a-zA-Z]([a-zA-Z0-9])*    {toupper(yytext);
                  if (is_keyword(yytext)!=-1)
 return (KW);
                 else
                  return(IDENTIFIER );
                  }
":="                        return (ASSIGNMENT );
'({NQUOTE}|'')+'            return (CHARACTER_STRING);
":"                         return (COLON);
","                         return (COMMA);
[0-9]+                      return (DIGSEQ);
"."                         return (DOT);
".."                        return (DOTDOT );
"="                         return (EQUAL);
">="                        return (GE);
">"                         return (GT);
"["                         return (LBRAC );
"<="                        return (LE);
"("                         return (LPAREN);
"<"                         return (LT);
"-"                         return (MINUS);
"<>"                        return (NOTEQUAL);
"+"                         return (PLUS);
"]"                         return (RBRAC);
[0-9]+"."[0-9]+             return (REALNUMBER);
")"                         return (RPAREN);
";"                         return (SEMICOLON);
"/"                         return (SLASH);
"*"                         return (STAR);
"**"                        return (STARSTAR);
"^"                         return (UPARROW);
"(*"                       do{ c=input();
                              if(c=='*'){
                        c=input();
                                 if (c==')')
                          break;
                        else unput(c);
                                      }
                             }while(1);
"{"                          while((c=input())!='}');
[ \n\t]
                             printf("ILLEGAL   ");
```

دقت کنید که در این برنامه آنچه برگردانده می‌شود رشته نیست بلکه نشانه است. همانطور که ملاحظه می‌گردد با کشف فضاهای خالی و یا توضیحات هیچ مقداری

بازگردانده نمی‌شود. به این ترتیب توضیحات و فضاهای خالی از دید تحلیلگر نحوی مخفی می‌ماند.

برای اینکه زبان مبدا، حساس به حالت نباشد کامپایلر باید حروف بزرگ و کوچک را یکسان در نظر بگیرد، به این منظور تحلیلگر لغوی باید حروف کوچک و بزرگ را یکسان در نظر بگیرد. یکی از روشهای پیاده سازی تحلیلگر لغوی به گونه‌ای که حساس به حالت نباشد در ذیل نشان داده شده است (تابع yywrap به پردازش آخر فایل کمک می کند.)

```
%{
#include <stdio.h>
int line_no = 1;
%}
A [aA]
B [bB]
C [cC]
D [dD]
E [eE]
F [fF]
G [gG]
H [hH]
I [iI]
J [jJ]
K [kK]
L [lL]
M [mM]
N [nN]
O [oO]
P [pP]
Q [qQ]
R [rR]
S [sS]
T [tT]
U [uU]
V [vV]
W [wW]
X [xX]
Y [yY]
Z [zZ]
NQUOTE [^']
%%
{A}{N}{D}                    return(AND);
{A}{R}{R}{A}{Y}              return(ARRAY);
{C}{A}{S}{E}                 return(CASE);
{C}{O}{N}{S}{T}              return(CONST);
{D}{I}{V}                    return(DIV);
{D}{O}                       return(DO);
{D}{O}{W}{N}{T}{O}           return(DOWNTO);
{E}{L}{S}{E}                 return(ELSE);
{E}{N}{D}                    return(END);
{E}{X}{T}{E}{R}{N} |
```

```
{E}{X}{T}{E}{R}{N}{A}{L}    return(EXTERNAL);
{F}{O}{R}                   return(FOR);
{F}{O}{R}{W}{A}{R}{D}       return(FORWARD);
{F}{U}{N}{C}{T}{I}{O}{N}    return(FUNCTION);
{G}{O}{T}{O}                return(GOTO);
{I}{F}                      return(IF);
{I}{N}                      return(IN);
{L}{A}{B}{E}{L}             return(LABEL);
{M}{O}{D}                   return(MOD);
{N}{I}{L}                   return(NIL);
{N}{O}{T}                   return(NOT);
{O}{F}                      return(OF);
{O}{R}                      return(OR);
{O}{T}{H}{E}{R}{W}{I}{S}{E} return(OTHERWISE);
{P}{A}{C}{K}{E}{D}          return(PACKED);
{B}{E}{G}{I}{N}             return(PBEGIN);
{F}{I}{L}{E}                return(PFILE);
{P}{R}{O}{C}{E}{D}{U}{R}{E} return(PROCEDURE);
{P}{R}{O}{G}{R}{A}{M}       return(PROGRAM);
{R}{E}{C}{O}{R}{D}          return(RECORD);
{R}{E}{P}{E}{A}{T}          return(REPEAT);
{S}{E}{T}                   return(SET);
{T}{H}{E}{N}                return(THEN);
{T}{O}                      return(TO);
{T}{Y}{P}{E}                return(TYPE);
{U}{N}{T}{I}{L}             return(UNTIL);
{V}{A}{R}                   return(VAR);
{W}{H}{I}{L}{E}             return(WHILE);
{W}{I}{T}{H}                return(WITH);
[a-zA-Z]([a-zA-Z0-9])+      return(IDENTIFIER);
":="                        return(ASSIGNMENT);
'({NQUOTE}|'')+'            return(CHARACTER_STRING);
":"                         return(COLON);
","                         return(COMMA);
[0-9]+                      return(DIGSEQ);
"."                         return(DOT);
".."                        return(DOTDOT);
"="                         return(EQUAL);
">="                        return(GE);
">"                         return(GT);
"["                         return(LBRAC);
"<="                        return(LE);
"("                         return(LPAREN);
"<"                         return(LT);
"-"                         return(MINUS);
"<>"                        return(NOTEQUAL);
"+"                         return(PLUS);
"]"                         return(RBRAC);
[0-9]+"."[0-9]+             return(REALNUMBER);
")"                         return(RPAREN);
";"                         return(SEMICOLON);
"/"                         return(SLASH);
"*"                         return(STAR);
"**"                        return(STARSTAR);
"->"  |
```

```
"^"                             return(UPARROW);
"(*"  |
"{"                             {
                                    register int c;
                                    while ((c = input()))
                                        {
                                            if (c == '}')
                                               break;
                                            else if (c == '*')
                                             {
                                                if ((c = input()) == ')')
                                                        break;
                                                else
                                                        unput (c);
                                            }
                                            else if (c == '\n')
                                            line_no++;
                                            else if (c == 0)
                                             commenteof();
                                             }
                                             }

[ \t\f]                             ;
\n                                  line_no++;
.       {
 fprintf (stderr, "'%c' (0%o): illegal
charcter at line %d\n", yytext[0],
yytext[0], line_no);
   }

%%

commenteof()
{
    fprintf (stderr, "unexpected EOF inside comment at line %d\n",line_no);
    exit (1);
}

yywrap ()
{
 return (1);
}
```

با توجه به آنچه در مورد پوشش خطا ذکر شد، می‌توان برخی از خطاهای متداول را مدیریت کرد. به عنوان مثال یکی از خطاهای متداول استفاده از شناسه‌ای است، که با رقم شروع می‌شود. می‌توان به مجموعه عبارات باقاعده، عبارت باقاعده‌ای به صورت ذیل اضافه کرد.

```
[0-9][a-zA-Z0-9]*          {printf("bad identifier");}
```

در این حالت اگر شناسه‌ای با رقم شروع گردد پیغام bad identifier چاپ شده و تحلیلگر به کار خود ادامه می‌دهد. در این شرائط پیغام خطا کاملا برای برنامه‌نویس گویا است و این خطا باعث توقف تحلیلگر لغوی نخواهد شد.

تمرینات

۱. برای هر یک از مجموعه های ذیل یک عبارت با قاعده ارائه دهید.

الف - مجموعه رشته هایی از 0 و1 به طوریکه با 0 شروع و تمام شوند.

ج - مجموعه رشته هایی از 0 و1 به طوریکه در این رشته هیچ دو یکی در کنار هم نباشند.

د - مجموعه رشته هایی از 0 و1 به طوریکه این رشته ها فقط شامل دو 1 باشند.

۲. اگر l1={1,12,22,00} و l2={11,111} باشد، زبانهای l1.l2 و *l1 را مشخص کنید.

۳. DFA مطابق با جدول تغییر حالت ذیل را رسم کنید، اگر A حالت شروع و D, F حالات نهایی باشند.

	a	b
A	E	B
B	C	D
C	A	G
D	C	B
E	F	G
F	C	D
G	C	E

٤

. برای عبارات باقاعده ذیل، NFA بسازید.

الف- (a|b)(a|b)

ب- (a|b)*(a|b)

ج- 0(10)*(01)

۵. برای عبارات با قاعده ذیل، NFA بسازید، سپس NFA ها را به DFA تبدیل کنید؟

الف- (a|b)(a|b)

ب- (a|b)*(a|b)

ج- 0(10)*(01)

٦. برای عبارات با قاعده ذیل، مستقیما DFA بسازید.

الف- (a|b)(a|b)

ب- (a|b)*(a|b)

ج- 0(10)*(01)

۷. برنامه ای به زبانC یا پاسکال بنویسید که رشته های تولیدی عبارت باقاعده (a|b)*(a|b)c را تشخیص دهد، راهنمایی: ابتدا DFA ی این عبارت با قاعده را رسم کرده وسپس برنامه آن را بنویسد.

۸ برنامه ای به زبان بهC یا پاسکال بنویسید که لغات زبان java را تشخیص داده و با کشف هر لغت پیغام مناسب را چاپ کند.

۹. با استفاده از flex یک تحلیلگر لغوی بسازید که لغات زبان جاوا را شناسایی کند.

۱۰. با استفاده از flex یک تحلیلگر لغوی بسازید که تگهای html را تشخیص دهد.

۱۱. با استفاده از flex یک تحلیلگر لغوی برای عبارات SQL بسازید.

۱۲. با استفاده از flex برنامه ای بنویسید که یک فایل متنی را بخواند و دنباله ای از چند فضای خالی را با یک فضای خالی جایگزین کند به طوریکه بین هر دو لغت فقط یک فضای خالی باشد.

۱۳. با مراجعه به مستندات flex عملکرد برنامه ذیل را تشریح کنید(yyin یک متغیر در محیط flex است که نوع ورودی(مانند فایل یا صفحه کلید) را مشخص می‌کند.)

```
%option noyywrap
int num_lines = 0, num_chars = 0;
 %%
\n          ++num_lines; ++num_chars;
 .          ++num_chars;
 %%
 main(int argc, char **argv) {
 ++argv, --argc;
if ( argc 0 )
       yyin = fopen( argv[0], "r" );
else
       yyin = stdin;
 yylex();
printf( "# of lines = %d, # of chars = %d\n", num_lines, num_chars );
}
```

۱٤. با مراجعه به مستندات flex عملکرد برنامه ذیل را تشریح کنید.

```
%{
 int mylineno = 0;
 void handle_comments(void);
%}
```

```
string      \"[^\n"]+\"
char        \'[^\n']+\'
prepro      ^#.*
ws          [ \t]+
alpha       [A-Za-z_]
dig         [0-9]
name        {alpha}({alpha}|{dig})*
int_num     [-+]?{dig}+
num1        [-+]?{dig}+\.?([eE][-+]?{dig}+)?
num2                [-+]?{dig}*\.{dig}+([eE][-+]?{dig}+)?
Flt_num             {num1}|{num2}
%%
{ws}
{prepro}
"//".*
"/*"                { handle_comments(); }
\n                  { mylineno++; }
{int_num}           { printf("integer number [%s] \n",yytext); }
{flt_num}           { printf("floating-point number [%s] \n",yytext); }
{name}              { printf("name [%s]\n",yytext); }
{string}            { printf("string [%s]\n",yytext); }
.                   { printf("don't recognise [%s]\n",yytext); }
%%
const char * keywords[] = {
   "auto","break","case","char","const","continue","default","do",
   "double", "else",    "enum", "extern","float", "for", "goto",
   "if", "int", "long", "register", "return", "short", "signed",     "sizeof", "static", "struct",
"switch", "typedef", "union", unsigned",
   "void", "volatile", "while", };
const int keywords_length = sizeof(keywords)/sizeof(keywords[0]);

int main( )
{
 printf("*** STARTING LEXICAL ANALYSIS ***\n");
 while(yylex() != 0);
 printf("*** FINISHED LEXICAL ANALYSIS ***\n");
 return 0;
}
void handle_comments(void)
{
 int c;
while((c = input()) != 0)    {
    if(c == '\n')
            ++mylineno;
    else if(c == '*')       {
            if((c = input()) == '/')
               break;
             else
               unput(c);
            }
   }
}
```

فصل سوم

تحلیلگر نحوی

هدف کلی

آشنایی با مفاهیم، تئوریها، تکنیکها، روشها و ابزارهای ساخت تحلیلگر نحوی

هدفهای رفتاری

پس از مطالعه این فصل انتظار می‌رود دانشجو بتواند:

۱- وظایف تحلیلگر نحوی را بیان کند.

۲- رابطه تحلیلگر نحوی و تحلیلگر لغوی را بیان کند.

۳- درخت تجزیه را تشریح کند.

٤- انواع روشهای تجزیه را بیان کند.

٥- خصوصیات تجزیه کننده بالا به پایین را بیان کند.

٦- خصوصیات تجزیه کننده پایین به بالا را بیان کند.

۷- تجزیه کننده پیشگو را شرح دهد.

۸- تجزیه کننده پیشگوی غیر بازگشتی را شرح دهد.

۹- تجزیه کننده LR(0) را شرح دهد.

۱۰- تجزیه کننده SLR(1) را شرح دهد.

۱۱- تجزیه کننده LR(1) را شرح دهد.

۱۲- تجزیه کننده LALR(1) را شرح دهد.

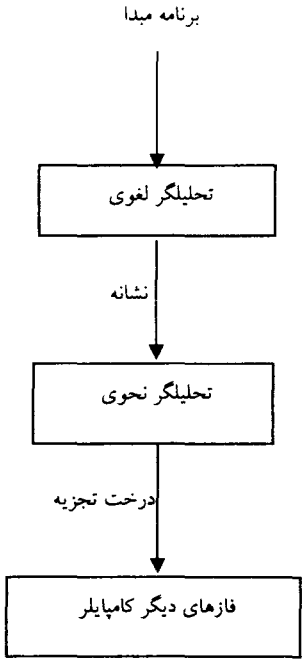
۱۳- نحوه ترجمه یک دنباله نشانه های ورودی را بر اساس گرامر شرح دهد.

۱٤- بوسیله تولید کننده تجزیه کننده YACC و یا Bison بتواند تجزیه کننده تولید کند.

۳-۱ مقدمه

در فصل گذشته ، تحلیلگر لغوی را مورد مطالعه و بررسی قرار دادیم و نشان دادیم که، تحلیلگر لغوی صحت ساختار تک تک لغات برنامه مبدا را بررسی می‌کند، اما صحت ترتیب لغات برنامه مبدا را بررسی نمی‌کند. بررسی صحت ترتیب لغات برنامه مبدا وظیفه تحلیلگر نحوی است.

تحلیلگر نحوی علاوه بر بررسی صحت ترتیب لغات برنامه مبدا، وظیفه ایجاد درخت تجزیه را برعهده دارد. ورودی تحلیلگر نحوی دنباله‌ای از نشانه‌ها است که توسط تحلیلگر لغوی تولید شده است و خروجی تحلیلگر نحوی درخت تجزیه است که نشان دهنده ساختار برنامه مبدا است. البته این درخت به شرطی ایجاد می‌شود که ساختار نحوی برنامه مبدا صحیح باشد. شکل ۳-۱ رابطه بین تحلیلگر نحوی و لغوی را نشان می‌دهد.



شکل۳-۱ رابطه تحلیلگر نحوی و لغوی

ساخت تحلیلگر نحوی نیازمند توصیف دقیق و کامل ساختار نحوی زبان مبدا است. ساختار نحوی زبانهای برنامه‌نویسی بوسیله قوانین بازگشتی توصیف می‌شوند. این قوانین، نشان دهنده ترتیب صحیح لغات برنامه است.

مثال ۳-۱ به دو قطعه برنامه ذیل که به زبان پاسکال نوشته شده اند، دقت کنید.

```
a- Program  sample;
   var   a:integer;

b- var a:integer;
   program  sample;
```

قطعه برنامه a و b از منظر تحلیلگر لغوی صحیح هستند. ترتیب لغات قطعه برنامه a مطابق قوانین نحوی زبان پاسکال است در حالیکه ترتیب لغات قطعه برنامه b مطابق قوانین نحوی زبان پاسکال نیست، در نتیجه تحلیلگر نحوی، برای قطعه برنامه b اعلام خطا خواهد کرد.

ساختار نحوی زبان را به روشهای مختلفی می‌توان بیان کرد، یکی از روشهای مهم بیان قوانین نحوی زبان مبدا، استفاده از گرامرهای مستقل از متن است. استفاده از گرامرهای مستقل از متن برای توصیف قوانین نحوی زبان مبدا دارای مزایای ذیل است:

۱- گرامرهای مستقل از متن، قوانین نحوی زبان مبدا را، کامل، دقیق و قابل فهم توصیف می‌کنند.

۲- با استفاده از گرامرهای مستقل از متن می‌توان ابزارها و نرم افزارهایی ایجاد کرد که گرامرهای مستقل از متن را دریافت کرده و تحلیلگر نحوی آن را به طور خودکار ایجاد کنند. تولید خودکار تحلیلگر نحوی شامل همه انواع گرامر مستقل از متن نمی‌شود بلکه گرامر مستقل از متن باید ویژگیهای خاصی داشته باشد تا تولید خودکار تحلیلگر نحوی امکان پذیر باشد.

۳-۲ مدیریت خطا

یکی از مواردیکه نیاز به بررسی دارد بروز خطا در برنامه‌ها است. خطاهای برنامه در سطوح مختلفی رخ می‌دهد. این سطوح عبارتند از:

۱- **خطاهای لغوی**: خطاهایی که مربوط به ساختار هر لغت در برنامه مبدا می‌باشد. این نوع خطا ها را تحلیلگر لغوی کشف می‌کند. وقتی دنباله ای از کارکترها با هیچ یک از عبارات با قاعده منطبق نگردد، خطای لغوی رخ داده است. برخی از این خطاها در زبان C عبارتند از:

- استفاده از کاراکترهای غیر مجاز مانند: $temp و s= a @ B

- تعریف نادرست نام‌ها یا شناسه ها مانند: 5temp
- تعریف نادرست ثوابت، مانند: 12.654.765

۲- **خطاهای نحوی**: خطاهایی که مربوط به ترتیب لغات برنامه مبدا است. این نوع خطاها را تحلیلگر نحوی کشف می‌کند. به عنوان مثال برخی از این نوع خطاها در زبان پاسکال عبارتند از:

- پرانتزهای نامتعادل مانند: a:=((a+b*)
- سمت چپ غیر مجاز مانند: (a+b):=2*b
- استفاده نادرست از عملگرها مانند:

```
if (a<b<c) then
    a:=2;
```

۳- **خطاهای معنایی**: علاوه بر قوانین حاکم بر ساختار لغات و نحو زبان قوانین دیگری وجود دارد که توسط عبارات باقاعده و گرامرهای مستقل از متن قابل بیان نیست ولی رعایت آنها الزامی است. اگر این قوانین رعایت نگردد خطا های معنایی رخ می‌دهد. این نوع خطاها را تحلیلگر معنایی کشف می‌کند. برخی از این نوع خطاها عبارتند از:

نوع داده ها: اگر عملوندهای یک عملگر ناسازگار باشند، مانند، جمع یک آرایه و یک تابع

جریان کنترل: مقصد دستورات کنترلی که کنترل را از محلی به محل دیگر منتقل می‌کنند موجود نباشد، به عنوان مثال دستور break; باعث می‌شود کنترل از یک حلقه مانند while یا for خارج گردد. اگر break درون هیچ حلقه‌ای نباشد مقصد break مشخص نیست.

مثال۳-۲ برنامه ذیل علی رغم اینکه از نظر لغوی و نحوی صحیح است ولی دارای خطای معنایی است، زیرا با اجرای break; مشخص نیست کنترل به کجا باید منتقل گردد.

```
int main(){
int a;
a=2;
break;
}
```

کنترل یکتایی: در برخی زبانها یک متغیر فقط یک بار می‌تواند تعریف گردد. به عنوان مثال در زبان پاسکال اگر یک متغیر دوبار تعریف شود، خطا رخ می‌دهد.

٤- **منطقی**: خطاهایی که مربوط به منطق برنامه است. کشف این خطا ها بر عهده برنامه نویس است. خطاهای منطقی ناشی از اشکال در الگوریتم مورد استفاده برنامه است. برخی از این خطاها عبارتند از:

- ایجاد حلقه بی نهایت در برنامه: مانند:

```
a:=10;
```

```
while(a>2)
begin
b=b+1;
c=c+1;
end;
```

- خطای منطق برنامه: مانند برنامه ذیل که قرار است اعداد زوج کمتر از ۱۰۰ را تولید کند ولی واضح است که چنین کاری انجام نمی‌دهد.

```
a:=1;
while(a<100) do
begin
  writeln(a);
  a:=a+2;
end;
```

مدیریت خطا، بخش مهمی در کامپایلر می‌باشد. کامپایلر باید خطاها را کشف کرده و به برنامه نویس اطلاع دهد. هر چه پیغامهای خطا دقیق تر باشد، رفع خطا ها سریعتر انجام می‌شود. پیغام خطا باید شامل دو قسمت باشد که عبارتند از:

- محل خطا
- راهنمایی جهت رفع خطا

در مورد خطاهای لغوی تعیین محل خطا امکان پذیر است، اما تعیین محل دقیق خطاهای نحوی به سادگی امکان پذیر نیست. به عنوان مثال به برنامه ذیل دقت کنید.

```
program f;
var i, j:integer;
begin
readln(i);
if i>12 then
begin
i:=13;
j:=14;
writeln(i+j);

if i=12 then
begin
i:=1;
j:=4;
writeln(i+j);
end;
end.
```

در برنامه، end از دستور if اول حذف شده است. کامپایلر بورلند پاسکال نسخه 7.0 خط آخر برنامه را به عنوان منبع خطا معرفی می‌کند و پیغام خطای ذیل را ارائه می‌دهد. که به معنی انتظار علامت ; است.

```
Error 85: ";" expected.
```

همانطور که مثال فوق نشان می‌دهد، پیغام و محل خطای معرفی شده از طرف کامپایلر نه تنها برنامه نویس را راهنمایی نمی‌کند بلکه او را گمراه می‌کند، در نتیجه برنامه نویس را از

درک و رفع اشکال اصلی دور می‌کند. علت پیغام گمراه کننده ناتوانی کامپایلر در درک یک برنامه غلط است.

مثال۳-۳ در برنامه بالا هر یک از حالات ذیل محتمل است:

-Begin بعد از if(i>12) اضافی است.

-End مربوط به if(i>12) حذف شده است.

-Begin بعد از if i=12 اضافی است.

کامپایلر قادر به انتخاب بین این سه حالت نیست، زیرا هدف برنامه نویس را نمی‌داند.

۳-۳ گرامرهای مستقل از متن

به دلیل اهمیت گرامر مستقل از متن در ساخت تحلیلگر نحوی، در این بخش به معرفی این گرامر می‌پردازیم. گرامر مستقل از متن که از این پس در این کتاب به اختصار گرامر می‌نامیم، ابزار مناسبی جهت توصیف ساختار نحوی زبانهای برنامه سازی است. گرامر مستقل از متن متشکل از مجموعه‌ای از قواعد تولید[1] و یک نماد شروع[2] است. هر قاعده تولید، یک ساختار نحوی را تعریف می‌کند که دارای نام است. هر قاعده تولید، دارای دو قسمت است. بخش سمت چپ که نام ساختار است و بخش سمت راست که تمام شکلهای ممکن ساختار را نشان می‌دهد. بخش سمت راست می‌تواند شامل نمادهای پایانه[3] و غیر پایانه[4] باشد. پایانه‌ها و غیر پایانه را نمادهای گرامر می‌نامیم. در نوشتن گرامرهای مستقل از متن اصول ذیل رعایت می‌شود تا درک آنها ساده تر گردد. این اصول عبارتند از:

۱- پایانه‌ها را به یکی از صورتهای ذیل نشان می‌دهیم.

- حروف کوچک مانند:a,b,c
- عملگرها و علائم مانند: *,-,+
- ارقام مانند: 0,1,2,3,4

۲- غیر پایانه را به یکی از صورتهای ذیل نشان می‌دهیم.

- حروف بزرگ مانند A,B,C
- اسامی با حروف مانند: expr,start

۳- برای نشان دادن دنباله‌ای از پایانه‌ها و غیر پایانه‌ها از حروف لاتین است، مانند: α,β ..

٤- سمت چپ اولین قاعده تولید به عنوان نماد شروع است.

---

1. Production Rules
2. Start Symbol
3. Terminal Symbol
4. Non Terminal

مثال۳-٤ گرامر ذیل رشته‌های محاسباتی را تولید می‌کند.

E→ E + T | E - T | T
T→ T * F | T / F | F
F→ id

در این گرامر E,T و F غیر پایانه و /,*,+,- و id پایانه و E نماد شروع است.

برای تشریح ساختار نحوی زبان برنامه‌نویسی از نوع خاصی از گرامر مستقل از متن به نام BNF[1] استفاده می‌گردد. BNF شبیه گرامر مستقل از متن است. ویژگی‌های BNF عبارتند از:

- سمت چپ و راست قانون با =:: از یکدیگر جدا می‌شوند.
- غیر پایانه ها درون < > قرار می‌گیرد.
- انتخابهای[2] مختلف یک قاعده تولید با علامت | جدا می‌شوند.

مثال۳-۵ BNF ذیل بخشی از قوانین نحوی زبان C را نشان می‌دهد.

```
<program> ::= <includes><declarations><main>
<includes> ::= #include <lib>
<lib> ::= iostream.h | math.h | ...
<declarations> ::= <single_declaration>; | <single_declaration>; <declarations>
<single_declaration> ::= <type> <identi.ers>
<type> ::= float | int | ...
<identi.ers> ::= <single_identi.er> | <single_identi.er>, <identi.ers>
<main> ::= main() { <declarations> <instructions> return(0) };
<instructions> ::= <single_instruction>; | <single_instruction>; <instructions>
                    <single_instruction> ::= <assignment> | <cin> | <cout> |
                    <if_statement> | <loop> | ...
<assignment> ::= <identi.er> = <expression> |
<identi.er>:: = ...
```

امروزه نوع دیگری از BNF به نام EBNF[3] استفاده می‌گردد. EBNF نسبت به BNF تفاوتهایی به شرح ذیل دارد.

- پایانه ها درون علامت " " قرار داده می‌شوند.
- نمادها و رشته های اختیاری درون علامت [ ] قرار داده می‌شوند.
- برای نشان دادن تکرار صفر یا بیشتر مانند عبارات باقاعده از * استفاده می‌گردد.
- برای نشان دادن تکرار یک یا بیشتر مانند عبارات باقاعده از + استفاده می‌گردد.
- قسمتهای تکراری را فاکتور گیری کرده و در علامت { } قرار می‌دهیم.

به عنوان نمونه به گرامر ذیل دقت کنید.

```
Identier ::= { letter | "_" } { letter | digit | "_" }*
```

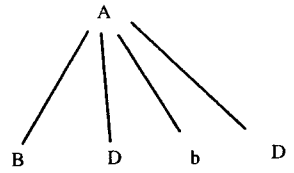
با توجه به اینکه BNF و EBNF نوعی گرامر مستقل از متن هستند در نتیجه در بخشهای بعدی گرامر های مستقل از متن را بررسی می‌کنیم.

---

1. Bakus Normal Form 2. Alternative 3. Extended BNF

۳-٤ درخت تجزیه

درخت تجزیه، چگونگی تولید یک رشته از نماد شروع گرامر را به صورت بصری نشان می‌دهد. در درخت تجزیه هر قاعده تولید مورد استفاده در تولید رشته، مانند A→BDbD به صورت یک گره نشان داده می‌شود به طوریکه که نام گره غیر پایانه سمت چپ قاعده تولید (یعنیA) و فرزندان گره، پایانه ها و غیر پایانه‌های سمت راست قاعده تولید (یعنی BDbD) هستند. به عنوان نمونه قاعده تولید A→BDbD به صورت ذیل نشان داده می‌شود.



شکل ۳-۲ گره معادل A→ BDbD

روابط ذیل بین درخت تجزیه و گرامر مستقل از متن برقرار است:

۱- ریشه درخت تجزیه متناظر با نماد شروع گرامر است.

۲- برگهای درخت تجزیه، پایانه گرامر یا ∈ است.

۳- گره داخلی درخت تجزیه متناظر با یک غیر پایانه است.

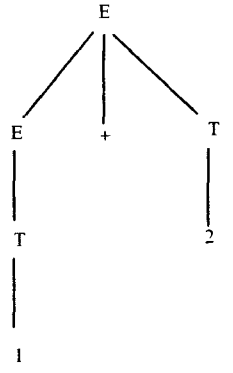
٤- اگر قاعده تولیدی به صورت A→∈ در گرامر باشد، در این صورت در درخت تجزیه گره A می‌تواند فرزند ∈ داشته باشد.

٥- اگر A غیر پایانه‌ای متناظر با یک گره داخلی درخت تجزیه باشد و ... ,X1,X2 برچسبهای فرزندان این گره از چپ به راست باشند آنگاه در گرامر می‌بایست قاعده تولیدی به صورت A→ X1,X2, ... وجود داشته باشد.

درخت تجزیه ساختار نحوی برنامه مبدا را نشان می‌دهد به همین جهت به درخت تجزیه، درخت نحوی نیز گفته می‌شود. برای هر رشته تولید شده توسط یک گرامر می‌توان حداقل یک درخت تجزیه تولید کرد.

مثال ۳-٦ درخت تجزیه رشته 1+2 با توجه به گرامر ذیل در شکل ۳-۳ نشان داده شده است.

E→ E+T | E-T | T
T→ 1 | 2 | 3



شکل۳-۳ درخت تجزیه رشته 2+1

اگر برگهای درخت را از چپ به راست بخوانیم رشته 2+1 به دست می‌آید.

۳-۵ اشتقاق

فرآیند تولید یک رشته از نماد شروع را اشتقاق می‌گوییم. تولید یک رشته بوسیله گرامر مستقل از متن از نماد شروع آغاز می‌شود. در هر مرحله یک غیر پایانه بوسیله سمت راست قاعده تولید آن گسترش داده می‌شود. این روند آنقدر تکرار می‌گردد تا هیچ غیر پایانه ای باقی نماند.

مثال ۳-۷ گرامر ذیل را در نظر می‌گیریم.

expr→ expr op expr
expr→ID
op→ +|-

به عنوان مثال رشته ID+ID توسط این گرامر به صورت ذیل تولید می‌گردد.

1- expr
2- expr op expr
3- ID op expr
4- ID + expr
5- ID + ID

هر دنباله ای از پایانه‌ها و غیر پایانه ها که در هر مرحله اشتقاق تولید می‌شوند را شبه جمله می‌نامیم. به عنوان مثال هر یک از دنباله های expr op expr و ID+expr شبه جمله هستند. در هر مرحله از فرآیند اشتقاق دو مسئله باید مد نظر قرار گیرد.

۱- اگر در شبه جمله چند غیر پایانه وجود داشته باشد باید یکی را برای گسترش انتخاب شود. به عنوان مثال در مرحله یک از مثال ۳-۷، سه غیر پایانه در دنباله expr op expr جهت گسترش وجود دارد که باید یکی انتخاب شود.

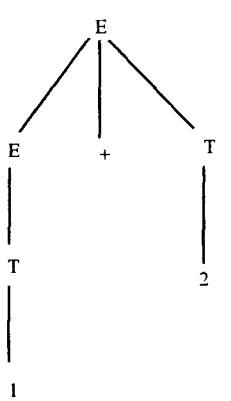
۲- اگر برای غیر پایانه انتخاب شده چند قاعده تولید وجود داشته باشد باید یکی انتخاب گردد. به عنوان مثال در ID op expr برای گسترش op دو قاعده تولید وجود دارد، که باید یکی انتخاب شود.

اگر در فرآیند اشتقاق در هر مرحله سمت چپ ترین غیر پایانه انتخاب شود، اشتقاق را سمت چپ ترین اشتقاق می‌نامیم و اگر در فرآیند اشتقاق در هر مرحله سمت راست ترین غیر پایانه انتخاب شود اشتقاق را سمت راست ترین اشتقاق می‌نامیم.

جدول ۳-۱ سمت چپ ترین و سمت راست ترین اشتقاق

سمت چپ ترین اشتقاق	سمت راست ترین اشتقاق
expr	expr
expr op expr	expr op expr
ID op expr	expr op ID
ID + expr	expr + ID
ID + ID	ID + ID

تفاوت درخت تجزیه و اشتقاق در این است که درخت تجزیه ترتیب گسترش غیر پایانه ها را نشان نمی‌دهد. به عنوان مثال در درخت تجزیه ذیل مشخص نیست، ابتدا E گسترش داده شده است یا T.



شکل ۳-٤ درخت تجزیه 1+2

اما در اشتقاق ترتیب گسترش غیر پایانه‌ها مهم است. به عنوان مثال دو اشتقاق ذیل با یکدیگر متفاوت هستند.

E	E
E+T	E+T
T+T	E+2
1+T	T+2
1+2	1+2

اگر سمت چپ‌ترین و سمت راست‌ترین اشتقاق را به صورت درخت تجزیه نشان دهیم درخت سمت چپ‌ترین اشتقاق به سمت چپ و درخت سمت راست‌ترین اشتقاق به سمت راست متمایل است (شکل ۳-۵). گرامرهای مستقل از متن دارای ویژگیهایی هستند که بر روی ساخت تحلیلگر نحوی تاثیر می‌گذارد. در ادامه به شرح مختصر برخی از این ویژگیها می‌پردازیم.

۳-۶ گرامرهای مبهم

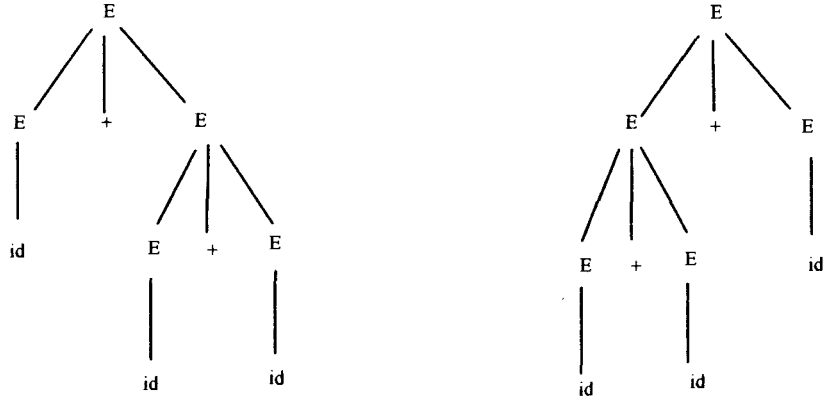
گرامر مبهم، گرامری است که بتوان برای رشته تولید شده از گرامر بیش از یک درخت تجزیه تولید کرد.

مثال۳-۸ گرامر ذیل و رشته id+id+id را در نظر می‌گیریم.

E→E+E|E-E|E

E→id

برای رشته id+id+id می‌توان دو درخت تجزیه به صورت ذیل ایجاد کرد. در نتیجه این گرامر مبهم است.



شکل ۳-۵ تولید دو درخت تجزیه برای id+id+id

به منظور رفع ابهام روشهای مختلفی وجود دارد که عبارتند از:

۱- تغییر گرامر: هر گرامری یک زبان را توصیف می‌کند. برای تولید یک زبان (مجموعه ای از رشته‌ها) می‌توان از گرامرهای متعددی استفاده کرد. در نتیجه گرامر مبهم را می‌توان تغییر داد که همان زبان را تولید کند و مبهم نیز نباشد.

مثال۳-۹ گرامر ذیل که برای تولید دستورات if بکار می‌روند را در نظر می‌گیریم.

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

به سادگی می‌توان بوسیله رشته if E1 then S1 else if E2 then S2 else S3 نشان داد که این گرامر مبهم است. به جای استفاده از گرامر فوق می‌توان از گرامر ذیل استفاده کرد که غیر مبهم است.

stmt → mathed | unmatched
matched → **if** expr **then** mathed **else** matched
| other
unmatched → **if** expr **then** stmt
| **if** expr **then** mathed **else** matched
| other

این گرامر برای هر رشته فقط یک درخت تجزیه تولید می‌کند. رفع ابهام به وسیله تغییر گرامر مشکلاتی دارد که عبارتند از:

- تبدیل یک گرامر مبهم به گرامر غیر مبهم، ممکن است باعث ایجاد گرامری گردد که درک آن و در نتیجه تولید تحلیلگر نحوی از آن مشکل‌تر باشد. به عنوان مثال گرامر غیر مبهم دستور if مثال ۳-۹ پیچیده‌تر از گرامر مبهم معادل آن است در نتیجه درک آن مشکل‌تر است.

- رفع ابهام بوسله تغییر گرامر همواره امکان پذیر نیست. ممکن است گرامر مبهمی وجود داشته باشد که گرامر معادل غیر مبهم نداشته باشد، چنین گرامری را گرامر ذاتا مبهم[1] می‌نامیم.

۲- استفاده از قوانین جانبی: در این روش گرامر را تغییر نمی‌دهیم اما قوانینی وضع می‌کنیم که به موجب آن از چند درخت تجزیه فقط یکی انتخاب شود. در این صورت می‌توان تصور کرد که گرامر برای هر رشته فقط یک درخت تجزیه تولید می‌کند. در نتیجه گرامر مبهم مانند گرامر غیر مبهم است. به عنوان نمونه می‌توان از قوانین ذیل استفاده کرد.

- استفاده از اولویت: در این روش به برخی از نشانه‌ها اولویت بالاتر داده می‌شود. به عنوان مثال در عبارات محاسباتی اولویت * از + بیشتر است.

---

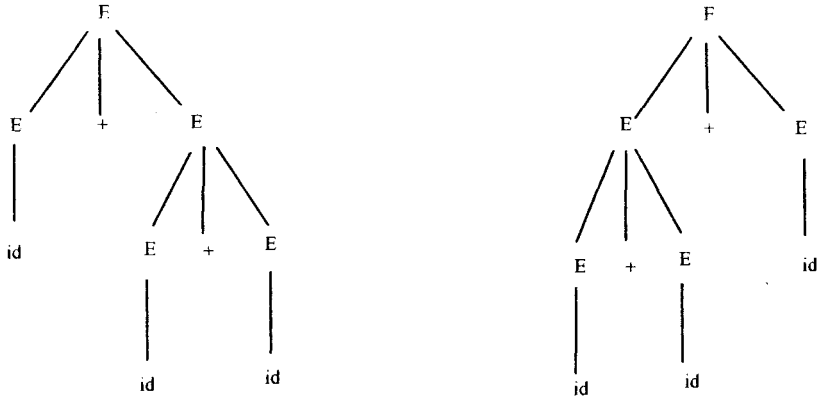
1. Inherently Ambiguous

- استفاده از شرکت پذیری: برای عملگرها می‌توان از شرکت پذیری چپ یا راست استفاده کرد. به عنوان مثال در عبارات محاسباتی + شرکت پذیری چپ دارد در نتیجه در عبارت a+b+c ابتدا a+b انجام می‌شود.

- استفاده از قوانین خاص: به عنوان مثال در گرامر مبهم دستور if، می‌توان از قانون ذیل استفاده کرد: هر else به نزدیکترین if مربوط می‌گردد.

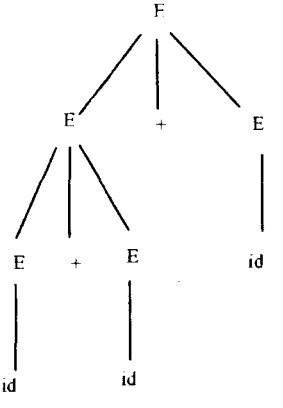
با استفاده از قوانین فوق بین چند درخت تجزیه فقط یکی انتخاب می‌گردد.

مثال ۳-۱۰ برای رشته id+id+id دو درخت تجزیه ذیل قابل ایجاد است.



شکل ۳-٦ تولید دو درخت تجزیه برای id+id+id

با توجه به اینکه+ شرکت پذیری چپ دارد، درخت تجزیه ذیل انتخاب می‌گردد.



شکل ۳-۷ تولید یک درخت تجزیه برای id+id+id

۳-۷ بازگشتی چپ

اگر سمت راست قاعده تولید با غیر پایانه سمت چپ قاعده تولید شروع شود، در این صورت گرامر دارای بازگشتی چپ است. به قواعد تولید ذیل دقت کنید.

$A \rightarrow A+T$

یا

$X \rightarrow XT$

غیر پایانه A دارای بازگشتی چپ است زیرا سمت راست قاعده تولید آن با غیرپایانه A شروع می‌شود. غیر پایانه X نیز دارای بازگشتی چپ است.

بازگشتی چپ به سه دسته تقسیم می‌شود که عبارتند از:

۱- بازگشتی چپ مستقیم: در این نوع بازگشتی، سمت راست قاعده تولید غیر پایانه با همان غیر پایانه شروع می‌شود.

مثال۳-۱۱ قاعده ذیل بازگشتی چپ مستقیم دارد.

$N \rightarrow N+T$

۲ - بازگشتی چپ غیر مستقیم: در این نوع بازگشتی، سمت راست قاعده تولید به طور غیر مستقیم با سمت چپ قاعده تولید شروع می‌گردد. به عنوان مثال غیرپایانه‌ای مانند A، مستقیما با A شروع نمی‌شود بلکه با غیرپایانه دیگری مانند B شروع می‌شود و سمت راست قاعده تولید غیر پایانه B با غیر پایانه A شروع می‌شود. این روند تا چند مرحله می‌تواند ادامه یابد و در نهایت دوباره به A می‌رسد.

مثال۳-۱۲ به گرامر ذیل دقت کنید.

$A \rightarrow Ba+a$
$B \rightarrow Cb$
$C \rightarrow A*a|c$

همانطور که ملاحظه می‌گردد غیر پایانه A با B و غیر پایانه B با C و در نهایت C دوباره با A شروع می‌گردد. در نتیجه به طور غیر مستقیم A با A شروع شده است، در نتیجه دارای بازگشتی چپ غیر مستقیم است.

۳- بازگشتی چپ مخفی: در این نوع بازگشتی، سمت راست قاعده تولید با سمت چپ شروع نمی‌شود ولی در شرائط خاص ممکن است سمت راست قاعده تولید با سمت چپ شروع شود. به عنوان مثال، سمت راست قاعده تولید $A \rightarrow BAD$ با A شروع نمی‌شود اما اگر B بتواند $\in$ را تولید کند (به عنوان مثال قاعده تولید $B \rightarrow \in$ موجود باشد)، می‌توان به جای B، $\in$ را در $A \rightarrow BAD$ قرار داد در نتیجه $A \rightarrow BAD$ به $A \rightarrow AD$ تبدیل می‌گردد که نشان دهنده بازگشتی چپ است.

بازگشتی چپ مخفی و غیر مستقیم را می‌توان با جایگزینی به بازگشتی چپ مستقیم تبدیل نمود.

مثال۳-۱۳ گرامر ذیل دارای بازگشتی چپ غیر مستقیم است.

N→AT
A→Na|a

در قاعده تولید N→AT ، با جایگذاری غیر پایانه A با قاعده تولید A→Na|a، گرامر مورد نظر به گرامر ذیل تبدیل می‌گردد که دارای بازگشتی چپ مستقیم است.

N→NaT|aT

مثال۳-۱٤ گرامر ذیل را در نظر می‌گیریم.

N→BNd | a
B→b|∈

این گرامر دارای بازگشتی چپ مخفی است، زیرا B می‌تواند ∈ را تولید کند که در این صورت قاعده تولید N→BNd | a به N→Nd | a تبدیل می‌گردد. به منظور حذف بازگشتی چپ مخفی با جایگذاری B با ∈ در قاعده N→BNd|a ، گرامر مورد نظر به گرامر ذیل تبدیل می‌گردد، که بازگشتی چپ مخفی به بازگشتی چپ مستقیم تبدیل شده است.

N→ND|BND| a
B→b

با توجه به موارد ذکر شده، همه انواع بازگشتی‌های چپ را می‌توان به بازگشتی چپ مستقیم تبدیل کرد، در نتیجه اگر روشی برای حذف بازگشتی چپ مستقیم ارائه شود همه انواع بازگشتی‌های چپ را می‌توان حذف نمود. در ذیل روشی برای حذف بازگشتی چپ مستقیم ارائه می‌دهیم. بازگشتی چپ مستقیم را به صورت کلی ذیل نشان می‌دهیم.

A→Aα|β

در این گرامر:

α: دنباله نمادهایی که بعد از A قرار دارد.

β: سایر انتخابهای قاعده تولید که با A شروع نمی‌شود.

این گرامر جملات ذیل را تولید می‌کند.

β
βα
βαα
βααα
βαααα
...

برای تولید این رشته‌ها می‌توان از دو قاعده ذیل که بازگشتی چپ ندارند نیز استفاده نمود.

A→ βR
R→ αR|∈

مثال۳-۱۵ گرامر ذیل را در نظر بگیرید

expr → expr + term | term

term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

در این گرامر expr بازگشتی چپ دارد. با توجه به روش ذکر شده برای حذف بازگشتی چپ جایگذاریهای ذیل را در نظر می‌گیریم.

A=expr
α = + term
β= term

پس از حذف بازگشتی چپ از گرامر فوق گرامر ذیل به دست می‌آید.

expr → term R | term
R → + term R |∈
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

۳-۸ فاکتور گیری چپ[1]

برای یک غیر پایانه ممکن است انتخابهای مختلفی وجود داشته باشد به طوریکه شروع یکسان داشته باشند. به عنوان مثال به گرامرهای ذیل دقت کنید.

A→ a B | aD

و یا

A→ cd B | cdg D

چنین گرامرهایی ساخت تجزیه کننده را مشکل می‌کنند. به عنوان مثال ، اگر نماد ورودیa باشد نمی‌توان تعیین کرد از کدام انتخاب A→aB|aD باید برای گسترش استفاده کرد، زیرا هر دو aبا شروع می‌شوند. گرامرهایی که فاقد چنین ویژگی باشند برای ساخت تحلیلگر نحوی مناسبترند. در نتیجه سعی می‌کنیم این ویژگی را حذف کنیم. برای حذف چنین خاصیتی از فاکتورگیری چپ استفاده می‌کنیم. با استفاده از فاکتورگیری چپ می‌توان گرامر جدیدی تولید کرد که همان زبان را تولیدکند و این مشکل را نداشته باشد، این تکنیک را فاکتورگیری چپ می‌نامیم، بدین منظور از روش ذیل استفاده می‌کنیم.

برای غیر پایانه‌ای مانند A که دارای چندین انتخاب است، طولانی‌ترین رشته مشترک ابتدای انتخابهای مختلف سمت راست قواعده تولید A را می‌یابیم. این قسمت مشترک را با α نشان می‌دهیم، و قسمت غیر مشترک را با β_i نشان می‌دهیم. در نتیجه قاعده تولید را می‌توان به صورت ذیل نشان داد:

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 ... | \alpha\beta_n$

α شروع مشترک

β_i: قسمت غیر مشترک

گرامر فوق را می‌توان به صورت ذیل نشان داد که همان زبان را تولید می‌کند:

$A \rightarrow \alpha R$

---

1. Left Factoring

$R \rightarrow \beta_1 | \beta_2 | \beta_3 | ..$

مثال۳-۱۶ گرامر ذیل را در نظر می‌گیریم.

A→ cd B | cdg D
B→bB | c
D→dD | e

با توجه به A→cdB|cdgD قسمت مشترک cd است در نتیجه:

α= cd
β_1=gD
β_2=B

گرامر را می‌توان به صورت ذیل نشان داد.

A→ cd R
R→ B | g D
B→bB | c
D→dD | e

این گرامر نیز همان زبان را تولید می‌کند.

۳-۹ رابطه تحلیلگر لغوی و تحلیلگر نحوی

تحلیلگر لغوی به وسیله یک تابع پیاده سازی می‌گردد، که توسط تحلیلگر نحوی فراخوانی می‌گردد. تحلیلگر لغوی در هر فراخوانی، برنامه مبدا را بررسی کرده یک نشانه را استخراج و به تحلیلگر نحوی تحویل می‌دهد. نشانه‌های بازگشتی از تحلیلگر لغوی، پایانه‌های گرامر مستقل از متن مورد استفاده تحلیلگر نحوی است.

تحلیلگر نحوی با توجه به گرامر، هر جایی که نیاز به پایانه باشد تحلیلگر لغوی را فراخوانی می‌کند، به این ترتیب از تحلیلگر لغوی درخواست نشانه بعدی می‌کند. اگر نشانه بازگشتی از تحلیلگر لغوی، همان پایانه مورد انتظار باشد تحلیلگر نحوی به کار خود ادامه می‌دهد در غیر این صورت خطا رخ داده است. به عنوان نمونه گرامر ذیل که نشان دهنده بخشی از گرامر زبان پاسکال را در نظر می‌گیریم.

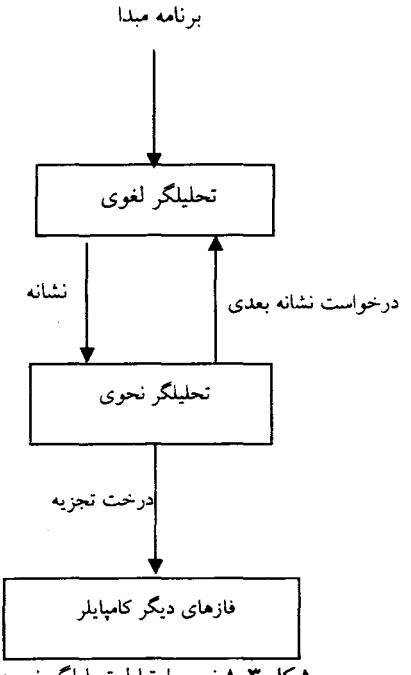
start→ PROGRAM ID SEMICOLON var_stmt block_stmt

بر اساس این گرامر تحلیلگر نحوی مراحل ذیل را انجام می‌دهد.

- ابتدا از تحلیلگر لغوی درخواست نشانه می‌کند. اگر تحلیلگر لغوی نشانه PROGRAM را برگرداند، تحلیلگر نحوی از تحلیلگر لغوی درخواست نشانه بعدی را می‌کند، در غیر این صورت خطا رخ داده است.
- تحلیلگر نحوی از تحلیلگر لغوی درخواست نشانه بعدی را می‌کند، اگر تحلیلگر لغوی نشانه ID را برگرداند، تحلیلگر نحوی از تحلیلگر لغوی درخواست نشانه بعدی را می‌کند. در غیر این صورت خطا رخ داده است.

- تحلیلگر نحوی از تحلیلگر لغوی درخواست نشانه بعدی را می‌کند، اگر تحلیلگر لغوی نشانه SEMICOLON را برگرداند، تحلیلگر نحوی از تحلیلگر لغوی درخواست نشانه بعدی می‌کند. در غیر این صورت خطا رخ داده است. ادامه روند بستگی به var_stmt دارد.

رابطه تحلیلگر لغوی و نحوی در شکل ۳-۸ نشان داده شده است.



شکل ۳-۸ نحوه ارتباط تحلیلگر نحوی و لغوی

۳-۱۰ تجزیه

تحلیلگر لغوی برنامه مبدا را به دنباله‌ای از نشانه‌ها تبدیل کرده و به تحلیلگر نحوی ارسال می‌کند. اگر این دنباله توسط گرامر زبان مبدا قابل تولید باشد، برنامه مبدا از نظر نحوی صحیح است در غیر این صورت دارای خطای نحوی است. فرآیندی لازم است که تعیین کند آیا دنباله‌ای از نشانه‌ها توسط گرامر زبان قابل تولید است، این فرآیند را تجزیه می‌نامیم. فرایند تجزیه نشان می‌دهد آیا دنباله‌ای از نشانه‌ها توسط گرامر قابل تولید است یا خیر. روالی که فرایند تجزیه را انجام می‌دهد، تجزیه کننده[1] نامیده می شود. تجزیه کننده‌ها به انواع مختلفی تقسیم می‌شوند، که در ذیل به بررسی آنها می‌پردازیم.

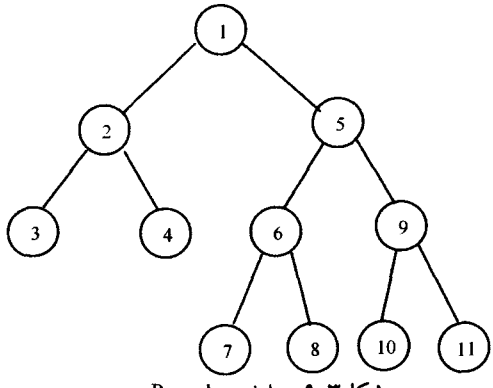
1. Parser

۳-۱۱ انواع تجزیه کننده‌ها

دو نوع تجزیه کننده وجود دارد که عبارتند از:

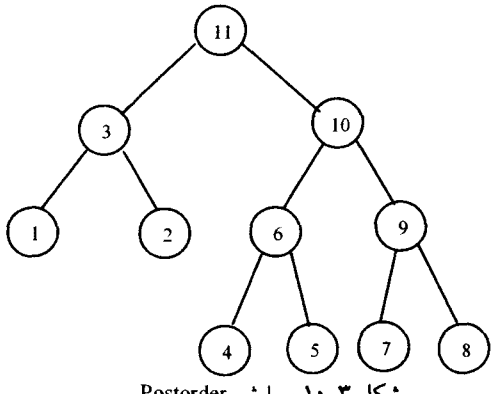
- تجزیه کنندههای بالا به پایین
- تجزیه کننده‌های پایین به بالا

همه تجزیه کننده‌ها با استفاده از گرامر برای رشته ورودی درخت تجزیه تولید می‌کنند. تفاوت آنها در ترتیب ایجاد گره‌های درخت تجزیه است. تجزیه کننده‌های بالا به پایین گره‌های درخت تجزیه را به ترتیبی که در پیمایش Preorder دیده می‌شود، می‌سازند. در حالیکه تجزیه کننده‌های پایین به بالا گره‌های درخت را به ترتیبی که در پیمایش Postorder دیده می‌شود می‌سازند. شکلهای ۳-۹ و ۳-۱۰ ترتیب ساخت گره‌ها را در هر دو تجزیه کننده نشان می‌دهد.



شکل۳-۹ پیمایش Preorder

ترتیب ساخت گره‌ها در تجزیه کننده‌های بالا به پایین



شکل ۳-۱۰ پیمایش Postorder

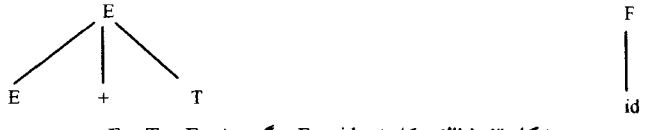
ترتیب ساخت گره‌ها در تجزیه کننده‌های پایین به بالا

همانطور که شکل نشان می‌دهد، در تجزیه کننده‌های بالا به پایین، یک گره ایجاد و سپس گره‌های فرزند آن از چپ به راست ایجاد می‌گردند، در تجزیه کننده پایین به بالا، ابتدا فرزندان گره و سپس خود گره ایجاد می‌شود.

تجزیه‌کننده‌های پایین به بالا چپ‌ترین گره‌ای که تمام فرزندان آن ساخته شده‌اند را ایجاد می‌کند. به عبارت دیگر تجزیه کننده‌های پایین به بالا ابتدا فرزندان یک گره را ایجاد می‌کنند و پس از ساختن تمام فرزندان گره، خود گره را ایجاد می‌کنند. به همین دلیل ریشه درخت تجزیه آخرین گره‌ای است که ایجاد می‌شود. تجزیه‌کننده‌های بالا به پایین و تجزیه‌کننده‌های پایین به بالا رشته ورودی را از چپ به راست پویش می‌کنند.

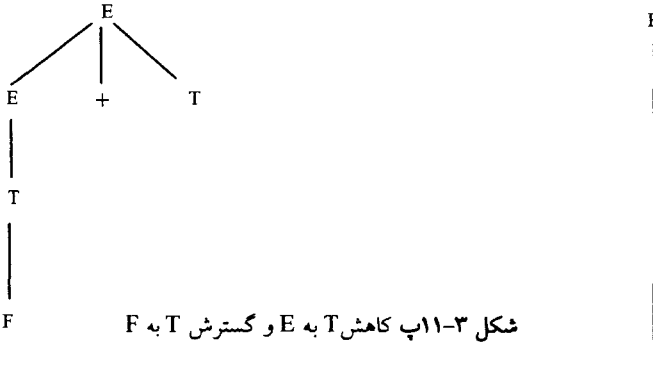
مثال ۳-۱۷ برای درک بهتر عملکرد دو نوع تجزیه کننده رشته id+id*id را با توجه به گرامر ذیل با هر دو روش تجزیه می‌کنیم.

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T / F \mid F$
$F \rightarrow id$

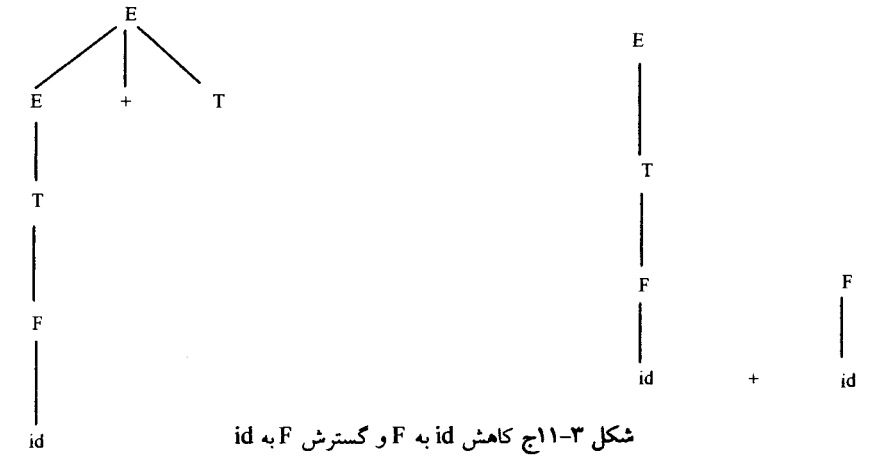


شکل ۳-۱۱الف کاهش id به F و گسترش E به E + T

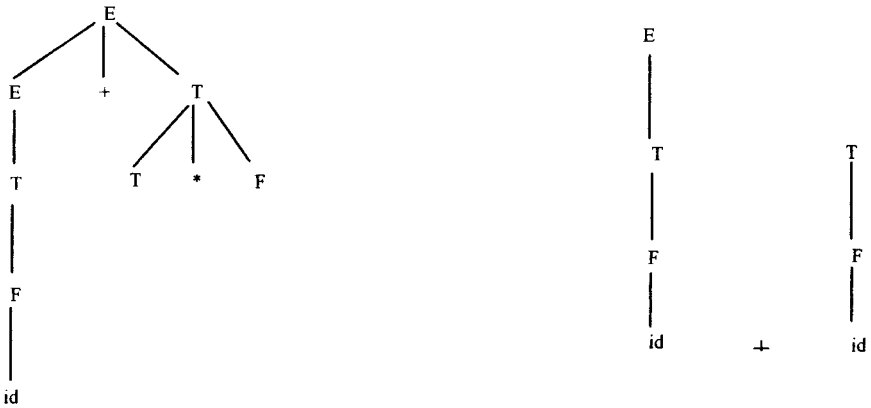
شکل ۳-۱۱ب کاهش F به T و گسترش E به T



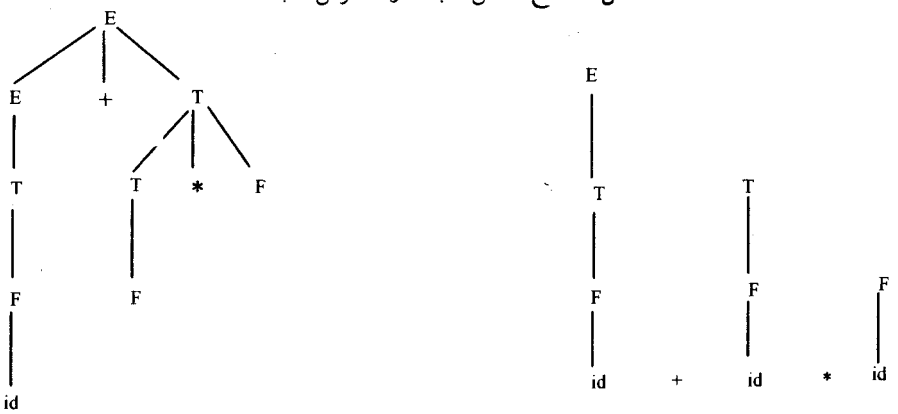
شکل ۳-۱۱پ کاهش T به E و گسترش T به F



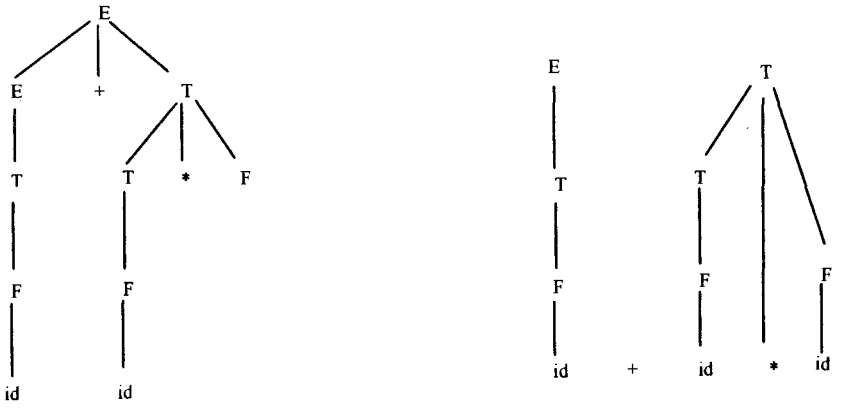
شکل ۳-۱۱ج کاهش id به F و گسترش F به id



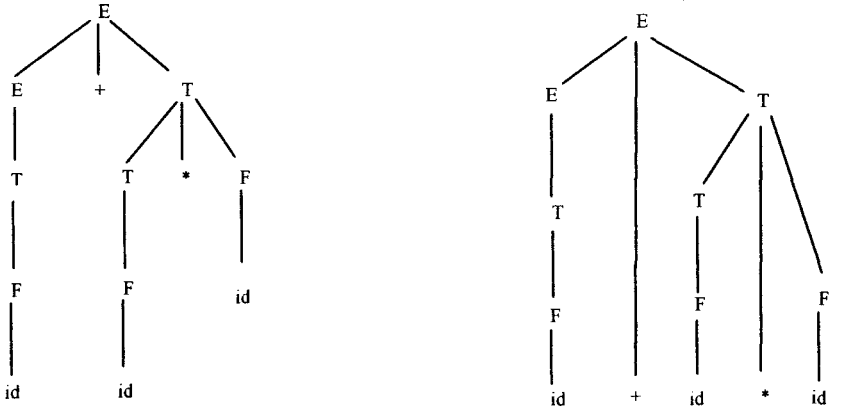
شکل ۳-۱۱ح کاهش F به T و گسترش T به T*F



شکل ۳-۱۱د کاهش id به F و گسترش T به F



شکل ۳-۱۱ ذ کاهش T*F به T و گسترش F به id



شکل ۳-۱۱ ر کاهش E+T به E و گسترش F به id

شکل ۳-۱۱ مراحل ساخت درخت تجزیه به روش پایین به بالا و بالا به پایین

همانطور که ملاحظه گردید هر دو تجزیه کننده درخت تجزیه یکسانی را ایجاد کردند، و تفاوت عملکرد آنها فقط در ترتیب ساخت گره‌های درخت تجزیه است. تجزیه کننده‌های بالا به پایین سمت چپ‌ترین غیر پایانه را گسترش می‌دهند. در نتیجه روش بالا به پایین سمت چپ‌ترین اشتقاق را ایجاد می‌کند.

اگر مراحل ایجاد درخت تجزیه توسط تجزیه کننده پایین به بالا را از آخر به اول مرور کنیم مشاهده می‌شود که سمت راست‌ترین اشتقاق تولید شده است. به عبارت دیگر، هر مرحله از تجزیه پایین به بالا عکس عمل سمت راست‌ترین اشتقاق است. بنابراین اولین گره

ای که در تجزیه پایین به بالا ایجاد می‌شود مطابق آخرین نمادی است که در سمت راست ترین اشتقاق گسترش می‌یابد.

تجزیه کننده بالا به پایین در هر مرحله یک غیر پایانه را با سمت راست قاعده تولید آن گسترش می‌دهند، تا در نهایت به رشته مورد نظر برسند. در حالیکه تجزیه کننده پایین به بالا در هر مرحله، سمت راست یک قاعده تولید را در دنباله ورودی یافته آن را به سمت چپ قاعده تولید کاهش می‌دهند تا در نهایت نماد شروع ایجاد شود.

تجزیه کننده‌های بالا به پایین، را می‌توان به روش دستی ایجاد کرد، یک برنامه‌نویس می‌تواند به سادگی یک تجزیه کننده بالا به پایین را ایجاد کند. در حالیکه تجزیه کننده‌های پایین به بالا، به حدی پیچیده است که ایجاد آن فقط با استفاده از ابزارهای تولید تجزیه کننده منطقی است.

به منظور تشریح نحوه ساخت تجزیه کننده‌های بالا به پایین و پایین به بالا دو مجموعه first و follow را تعریف می‌کنیم. درک این دو تعریف برای درک چگونگی ساخت تجزیه کننده‌ها بسیار ضروری است.

first(α)

اگر α دنباله‌ای از پایانه‌ها و غیر پایانه‌ها باشد، مجموعه first مربوط به α پایانه‌هایی را مشخص می‌کند که رشته‌های مشتق شده از α با آنها شروع می‌شوند. این مجموعه را با first(α) نشان می‌دهیم. اگر α بتواند ∈ را تولید کند، ∈ نیز به first(α) اضافه می‌شود.

مثال۳-۱۸ با توجه به گرامر ذیل first(BCd) را محاسبه کنید.

A→ BCd
B→ bB | e | ∈
C→ aC | ∈

به رشته‌هایی که از BCd مشتق شده‌اند دقت کنید.

BCd→**d**

یا

BCd→**b**BCd
bBd
bd

یا

BCd→**e**Cd
ed

یا

BCd→Cd
aCd
ad

با توجه به رشته‌های قابل اشتقاق از BCd ، ملاحظه می‌شود که شروع تمام رشته‌های مشتق شده نمادهای a,e,b و d قرار دارد بنابراین first(BCd)={d,b,e,a} است، با توجه به اینکه BCd نمی‌تواند ∈ را تولید کند در نتیجه ∈ در first(BCd) قرار نمی‌گیرد.

مثال۳-۱۹ با توجه به گرامر ذیل first(aA) و first(aB) را محاسبه کنید.

A→ a A|a B
B→ b B| c

با توجه به گرامر، رشته‌های مشتق شده از aA فقط با a و رشته‌های مشتق شده از aB فقط با a شروع می‌شوند، بنابراین:

first(aA) = {a }
first(aB) = {a }

مثال۳-۲۰ با توجه به گرامر زیر first(A) را مشخص کنید.

A→ aA | bB | a
B → bB | b | ∈

در این گرامر first(A)={a,b} زیرا تمام رشته‌هایی که از A مشتق می‌شوند، با پایانه‌های a یا b شروع می‌شوند. به منظور روشن تر شدن موضوع به تولید رشته‌هایی از A در ذیل دقت کنید.

A→ **a**

یا

A→ **a** A
aaA
aa...

یا

A→ **b**B
bbB
bb..

همانطور که ملاحظه می‌گردد تمام رشته‌ها با a یا b شروع می‌شوند. در نتیجه:

first(A)={a,b}

A نمی‌تواند ∈ را تولید کند در نتیجه ∈ به مجموعه first(A) اضافه نگردیده است.

مثال۳-۲۱ با توجه به گرامر مثال قبل، first(B)={b,∈} است. به منظور روشن‌تر شدن موضوع به تولید رشته‌هایی ازB در ذیل دقت کنید.

B → **b**
B → **b** B
bbB
bb...
B → **∈**

همانطور که ملاحظه می‌گردد تمام رشته‌ها با b شروع می‌شوند. از آنجاییکه B می‌تواند ∈ را تولید کند در نتیجه ∈ نیز به مجموعه first(B) اضافه می‌گردد، بنابراین:

first(B)={b, ∈}

مثال۳-۲۲ با توجه به گرامر first(bB) را محاسبه کنید.

با توجه به گرامر تمام رشته‌هایی که از bB مشتق می‌شوند با b شروع می‌شوند. در نتیجه:

first(bB)={b}

به منظور محاسبه first می‌توان از قوانین ذیل استفاده کرد.

۱- اگر α پایانه باشد آنگاه first(α) برابر مجموعه {α} است.

مثال۳-۲۳ در گرامر فوق داریم:

first(a)={a}
first(b)={b}

۲- اگر α بتواند ∈ را تولید کند، آنگاه ∈ به مجموعه first(α) اضافه می‌گردد.

۳- اگر X غیر پایانه و $X \rightarrow Y_1Y_2Y_3...Y_n$ باشد و مجموعه‌های $first(Y_1),first(Y_2),...first(Y_n)$ شامل ∈ باشند یعنی همه Y_i بتوانند تهی را تولید کنند. در نتیجه X نیز می‌تواند ∈ را تولید کند که در این صورت ∈ به مجموعه first(X) اضافه می‌گردد.

مثال۳-۲۴

$S \rightarrow AB$
$A \rightarrow aA \mid bB \mid a \mid \in$
$B \rightarrow bB \mid b \mid \in$

در $S \rightarrow AB$ غیر پایانه A نقش Y_1 و B نقش Y_2 را دارد. با توجه به گرامر مجموعه first(A) و first(B) شامل ∈ هستند و با توجه به $S \rightarrow AB$ در نتیجه S نیز می‌تواند ∈ را تولید کند، در نتیجه ∈ به مجموعه first(S) نیز اضافه می‌گردد.

۴- اگر X غیر پایانه و $X \rightarrow Y_1Y_2Y_3...Y_n$ باشد، مجموعه $first(Y_1)$ (به جز ∈)به مجموعه first(X) اضافه می‌گردد. زیرا $first(Y_1)$ مجموعه پایانه‌هایی هستند که در شروع رشته‌هایی که توسط Y_1 تولید می‌شوند قرار دارند از آنجاییکه X با Y_1 شروع می‌شود، پس X با پایانه‌های $first(Y_1)$ شروع می‌شوند، در نتیجه $first(Y_1)$ به first(X) اضافه می‌گردد.

مثال۳-۲۵ به گرامر ذیل دقت کنید

$A \rightarrow B\ ab$
$B \rightarrow c \mid d$

در این گرامر first(B)={c,d} است. با توجه به اینکه A با B شروع می‌شود بنابراین first(A) شامل مجموعه first(B) نیز می‌شود. در نتیجه :

first(A)={c,d}

۵- اگر X غیر پایانه و $X \rightarrow Y_1Y_2Y_3...Y_n$ باشد و ∈ در مجموعه $first(Y_1)$ باشد (Y_1 می‌تواند ∈ را تولید کند) در این صورت علاوه بر $first(Y_1)$ (به جز∈) مجموعه $first(Y_2)$ (به جز∈) نیز به first(X) اضافه می‌گردد.

با توجه به قانون ٤ ، $first(Y_1)$ به $first(X)$ اضافه می‌گردد. از آنجاییکه $first(Y_1)$ شامل $\in$ است پس Y_1 می‌تواند تهی شود. اگر Y_1 را تهی در نظر بگیریم، آنگاه $X \rightarrow Y_1Y_2Y_3...Y_n$ به $X \rightarrow Y_2Y_3...Y_n$ تبدیل می‌گردد. با توجه به قانون ٤ مجموعه $first(Y_2)$ نیز به مجموعه $first(X)$ اضافه می‌گردد.

مثال۳-۲٦ به گرامر ذیل دقت کنید

$A \rightarrow Bab$
$B \rightarrow c \mid d \mid \in$

در این گرامر $first(B)=\{c,d,\in\}$ است. با توجه به قانون ٤، A با B شروع می‌شود پس $first(A)$ شامل مجموعه $fisrt(B)=\{c,d, \in\}$ به جز $\in$ نیز می‌شود. از آنجاییکه $\in$ در مجموعه $first(B)$ قرار دارد. پس B می‌تواند تهی شود. در نتیجه $A \rightarrow Bab$ به $A \rightarrow ab$ تبدیل می‌شود. در $A \rightarrow ab$ طبق قانون ٤ ، غیر پایانه A با a شروع شده است در نتیجه $first(a)$ به مجموعه $first(A)$ اضافه می‌شود. با توجه به $first(a)=\{a\}$ نتیجه ذیل به دست می‌آید.

$first(A)=\{c,d,a\}$

با توجه به قوانین ٤ و ٥ اگر $X \rightarrow Y_1Y_2Y_3...Y_n$ و $\in$ در $first(Y_1)$ و $first(Y_2)$ باشد، مجموعه $first(Y_3)$ به جز $\in$ به مجموعه $first(X)$ اضافه می‌گردد.

با توجه به موارد ذکر شده می‌توان قانون کلی ذیل را ارائه داد:

با توجه به قاعده تولید $X \rightarrow Y_1Y_2Y_3...Y_i....Y_n$ اگر، $\in$ در $first(Y_1),first(Y_2),...first(Y_{i-1})$ باشد، مجموعه $first(Y_i)$ (به جز $\in$) به $fiirst(X)$ اضافه می‌شود.

follow(A)

مجموعه follow(A) پایانه‌هایی را مشخص می‌کند که در اشتقاق‌های مختلف بلافاصله در سمت راست A قرار می‌گیرند این مجموعه را با follow(A) نشان می دهیم.

مثال۳-۲۷ با توجه به گرامر ذیل follow(A) را محاسبه کنید.

$X \rightarrow aXAad \mid a$
$A \rightarrow Ac \mid Af \mid \in$

برای مشخص کردن follow(A) باید تمام شبه جملات قابل تولید از A در نظر گرفته شوند. به عنوان مثال چند شبه جمله را بررسی می‌کنیم.

$X \rightarrow aXAad$

در این شبه جمله پایانه a بلافاصله سمت راست A قرار دارد در نتیجه a به مجموعه follow(A) اضافه می‌شود. با جایگزینی‌های دیگر شبه جمله ذیل به دست می‌آید.

$X \rightarrow aXAad \rightarrow aXAcad$

در شبه جمله aXAcad پایانه c بلافاصله بعد از A قرار دارد در نتیجه c نیز در مجموعه follow(A) قرار دارد. با جایگزینی‌های دیگر، به شبه جمله ذیل می‌رسیم.

X→aXAad →aXAcad→ aXAfcad

در شبه جمله aXAfcad پایانه f بلافاصله بعد از A قرار دارد در نتیجه f نیز در مجموعه follow(A) قرار دارد. با توجه به مراحل بالا می‌توان نتیجه گرفت که:

follow(A)={a,c,f}

همانطور که ملاحظه گردید محاسبه follow نیاز به تولید تمام شبه جملات ممکن که شامل A باشند دارد. انجام چنین کاری بسیار زمانبر است. می‌توان از قوانین ذیل به منظور محاسبه follow استفاده کرد تا سریعتر به جواب رسید.

۱- اگر S نماد شروع باشد $ ($ نشان دهنده آخر رشته ورودی است) به follow(S) اضافه می‌شود.

زیرا در تجزیه کننده به آخر رشته ورودی $ اضافه می‌شود، در نتیجه رشتهw به w$ تبدیل می‌گردد. اگر w به غیر پایانه شروع کاهش یابد w$ به S$ تبدیل می‌گردد که $ بلافاصله سمت راست S قرار دارد، در نتیجه $ عضوی از follow(S) است.

۲- برای هر قاعده تولیدی به صورت M→αNβ ، تمام پایانه‌های موجود در first(β) (به جز ∈) به follow(N) اضافه می‌شود. زیرا first(β) مجموعه پایانه‌هایی است که در شروع β قرار دارد و با توجه به اینکه β در سمت راست N است بنابراین این پایانه‌ها در سمت راست N قرار می‌گیرند. در نتیجه باید عضو follow(N) باشند.

مثال۳-۲۸ به گرامر ذیل دقت کنید.

A→ AXZ | ∈
Z → aZ | bZ | c |∈
X→ a

در این گرامر first(Z)={a,b,c,∈} است. مطابق قانون ذکر شده مجموعه first(Z) به جز ∈ به follow(X) اضافه می‌شود. زیرا این مجموعه در شروع رشته‌های تولیدی Z قرار می‌گیرند، و Z در سمت راست X است در نتیجه مجموعه {a,b,c} (به جز ∈) به follow(X) اضافه می‌شود.

۳- برای هر قاعده ای به صورت M→αN و یا M→αNβ در صورتیکه ∈ عضو first(β) باشد یعنی β بتواند رشته تهی یا ∈ را تولید کند، تمام پایانه‌های مجموعه follow(M) به follow(N) اضافه می‌شود. زیرا follow(M) مجموعه پایانه‌هایی است که در یک شبه جمله مانند XMY در سمت راست M ظاهر می‌شود. اگر به جای M طبق قاعده تولید M→αN عبارت αN را جایگزین کنیم. XMY به XαNY تبدیل می‌شود، در نتیجه هر آنچه در سمت راست M (یعنی مجموعه follow(M)) است، در سمت راست N (یعنی follow(N)) قرار می‌گیرد.

همچنین اگر در XMY به جای M دنباله αNβ قرار گیرد دنباله XαNβY به دست می‌آید، با توجه به اینکه β می‌تواند تهی گردد، XαNβY به XαNY تبدیل می‌گردد. در نتیجه هر آنچه در سمت راست M (یعنی مجموعه follow(M)) است، در سمت راست N(یعنی follow(N)) قرار می‌گیرد.

مثال۳-۲۹ با توجه به گرامر ذیل follow(B) را محاسبه کنید.

A→AXb
X→d|dB|eBE
E→a|∈
B→b

- با توجه به AXb و با استفاده از قانون ۲ ، پایانه‌های first(b) به follow(X) اضافه می‌گردد. با توجه به first(b)={b} پایانه b به follow(X) اضافه می‌گردد.

- با توجه به eBE و با استفاده از قانون ۲ ، پایانه‌های first(E) به follow(B) اضافه می‌گردد. با توجه به first(E)={a,∈} ، a (مجموعه first(E) به جز ∈) به follow(B) اضافه می‌شود. همچنین با توجه به X→dB و follow(X)={b} و با استفاده از قانون ۳ ، پایانه‌های follow(X) به follow(B) اضافه می‌گردد. در نتیجه b به follow(B) اضافه می‌گردد، یعنی:

follow(B)={a,b}

مثال۳-۳۰ با توجه به گرامر ذیل follow(A) را محاسبه کنید.

S→AB
A→a
B→c|e|D
D→dD|d

با توجه به شبه جملات زیر نمادهای c,e,d می‌توانند بلافاصله بعد از A قرار گیرند.

AB→A**c**
AB→A**e**
AB→AD→A**d**D

در نتیجه با توجه به گرامر follow(A)={c,e,d} است.

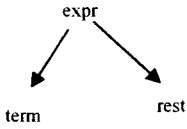
۳-۱۲تجزیه کننده بالا به پایین

در این بخش به شرح کامل تجزیه کننده بالا به پایین می‌پردازیم. گرامر ذیل را در نظر می‌گیریم:

expr→ term rest
term→ ID
rest →'+' expr | '-' expr | ∈

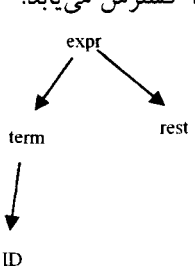
می‌خواهیم رشته ID+ID را با توجه به گرامر فوق تجزیه کنیم. تجزیه بالا به پایین از نماد شروع گرامر شروع می‌شود. از یک متغیر پیشنگر(lookahead) برای تشریح این روش استفاده می‌کنیم، این متغیر حاوی نشانه جاری در پویش چپ به راست رشته ورودی است.

به همین جهت در رشته ID+ID ابتدا lookahead=ID است زیرا اولین نشانه در پویش چپ به راست رشته ID+ID نشانه ID است. تجزیه کننده بالا به پایین ابتدا غیر پایانه شروع را گسترش می‌دهد، در نتیجه درخت تجزیه به صورت شکل ۳-۱۲ می‌شود.



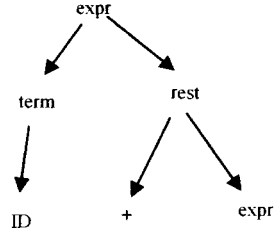
شکل ۳-۱۲ گسترش غیر پایانه شروع

سمت چپ ترین غیر پایانه درخت تجزیه یعنی term را در نظر می‌گیریم. term را در صورت امکان باید به گونه‌ای گسترش دهیم که نشانه‌ای که lookahead به آن اشاره می‌کند یعنی ID را تولید کند. به همین جهت از term→ID برای گسترش term استفاده می‌کنیم. درخت تجزیه به صورت شکل ۳-۱۳ گسترش می‌یابد.



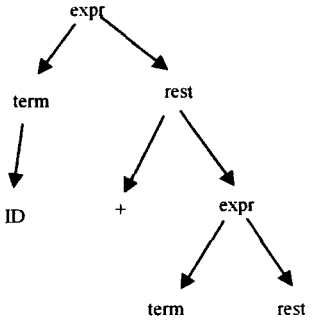
شکل ۳-۱۳ گسترش term

اکنون ID توسط درخت تجزیه تولید شده است که مطابق با lookahead=ID است در نتیجه lookahead در رشته ورودی به سمت راست حرکت می‌کند. lookahead شامل دومین نشانه دنباله ورودی یعنی lookahead='+' می‌شود. در مرحله بعد سمت چپ‌ترین غیر پایانه درخت تجزیه را انتخاب می‌کنیم. در این وضعیت تجزیه کننده، درخت تجزیه را باید به گونه‌ای گسترش دهد که+=lookahead را تولید کند، در نتیجه درخت تجزیه به صورت شکل ۳-۱۴ گسترش می‌یابد.



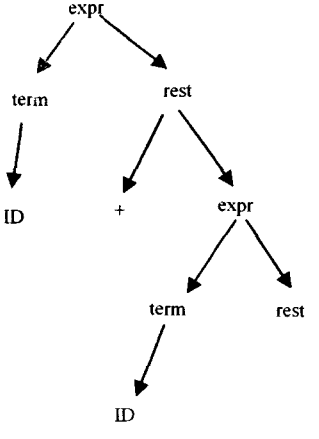
شکل ۳-۱۴ گسترش rest

در این وضعیت نشانه + توسط درخت تولید گردیده است، در نتیجه lookahead به نشانه بعدی یعنی lookahead=ID اشاره می‌کند. سمت چپ‌ترین غیر پایانه یعنی expr را انتخاب می‌کنیم. expr را به گونه‌ای گسترش می‌دهیم که نشانه‌ای را که lookahead به آن اشاره می‌کند یعنی ID را تولید کند. در این وضعیت از expr→ term rest استفاده می‌کنیم. درخت تجزیه را به صورت شکل ۳-۱۵ گسترش می‌دهیم.



شکل ۳-۱۵ گسترش expr

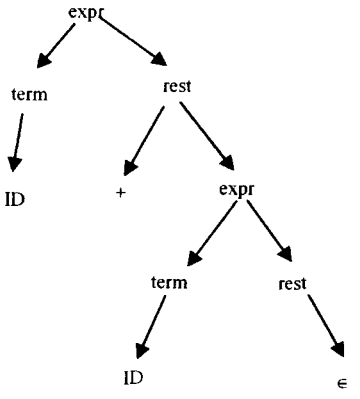
در این وضعیت غیر پایانه term را باید به گونه‌ای گسترش دهیم تا نشانه مورد نظر lookahead یعنی ID را تولید کند. در نتیجه درخت تجزیه شکل ۳-۱۶به دست می‌آید.



شکل ۳-۱۶ گسترش term

اکنون ID تولید شده است در نتیجه lookahed به نشانه بعدی اشاره می‌کند چون رشته تمام شده است lookahead خالی است. در این وضعیت ID+ID تولید گردیده است، اما

درخت هنوز شامل غیر پایانه rest است، چون کل رشته ورودی پویش گردیده است و lookahead به هیچ نشانه ای اشاره نمی‌کند. در نتیجه از rest→∈ استفاده می‌کنیم. درخت تجزیه به صورت شکل ۳-۱۷ تغییر می‌کند.



شکل ۳-۱۷ گسترش rest

همانطور که ملاحظه می شود در هر مرحله یک غیر پایانه را گسترش می‌دهیم. برای تعیین قاعده تولید مناسب از lookahead استفاده می‌شود. تجزیه کننده‌های بالا به پایین مختلفی وجود دارد که عبارتند از:

- تجزیه کننده پیشگو

تجزیه کننده پیشگو نوع خاصی از تجزیه کننده بازگشتی کاهشی[1] است. در تجزیه کننده بازگشتی کاهشی اجرای توابع بازگشتی، ورودی را پردازش می‌کنند. تجزیه کننده بازگشتی کاهشی از روش سعی و خطا برای پردازش ورودی استفاده می‌کند. یعنی تابعی برای پردازش ورودی فراخوانی می‌شود اگر پردازش ورودی شکست خورد به عقب بازگشته و تابع دیگری فراخوانی می‌شود. طبیعی است که این روش بسیار کند و زمانبر است. در تجزیه کننده پیشگو با توجه به نماد بعدی، تابعی که باید اجرا شود دقیقا مشخص می‌گردد و به همین جهت سریعتر است.

- تجزیه کننده پیشگوی غیر بازگشتی

در این تجزیه کننده با استفاده از پشته و جدول تجزیه، بدون فراخوانی بازگشتی توابع رشته ورودی پردازش می‌شود.

---

1. Recursive Descent Parser

۳-۱۳ تجزیه کننده پیشگو

در تجزیه کننده پیشگو، توابع بازگشتی وظیفه پردازش رشته ورودی را بر عهده دارند. برای هر غیر پایانه، یک تابع ایجاد می‌شود. در تجزیه کننده پیشگو بر اساس نماد ورودی بعدی، سمت راست مناسب غیر پایانه انتخاب می‌شود. به همین دلیل این تجزیه کننده را پیشگو می‌نامیم.

گرامر ذیل را در نظر بگیریم.

```
stmt→ IF expr THEN stmt
      | WHILE expr DO
      | REPEAT stmt UNTIL expr
      | FOR expr TO expr DO stmt
```

این گرامر ساختار برخی دستورات کنترلی زبان پاسکال را توصیف می‌کند. غیر پایانه stmt به صورت یک تابع به نام ()stmt پیاده‌سازی می‌گردد که وظیفه پردازش انتخابهای مختلف مربوط به stmt را بر عهده دارد. در تابع stmt برای انتخاب یکی از دستورات IF, WHILE,REPEAT و FOR نماد جاری در lookahead با IF, WHILE,REPEAT و FOR مقایسه می‌شود. اگر نماد lookahead نشانه IF باشد IF expr THEN stmt انتخاب می‌گردد و اگر نماد lookahead نشانه WHILE باشد WHILE expr DO انتخاب می‌گردد و بقیه موارد، به همین ترتیب انجام می‌گیرد.

به عنوان مثال دیگر گرامر ذیل را در نظر می‌گیریم این گرامر عبارات محاسباتی ساده را تولید می‌کند. برای این گرامر سه تابع ()term,expr و ()rest لازم است. زیرا سه غیر پایانه expr,term و rest موجود است.

```
expr→ term rest
term→ 1 | 2 | 3
rest→'+' expr | '-' expr | '*' expr | '/' expr | ∈
```

برای سادگی پیاده‌سازی فرض می‌کنیم رشته ورودی از صفحه کلید خوانده می‌شود. lookahead یک متغیر عمومی از نوع char است که حاوی نماد جاری ورودی است و برای اشاره lookahead به نماد ورودی بعدی از lookahead=getche(); استفاده می‌شود. getche(); یک کاراکتر را از ورودی می‌خواند. اگر بخواهیم تجزیه کننده محتوای فایل را تجزیه کند، از getc و یا fgetc به جای ()getche استفاده می‌شود.

از یک تابع کمکی به نام ()match استفاده می‌کنیم. این تابع نشانه جاری رشته ورودی که در lookahead است را با نماد مورد انتظار تجزیه کننده که با symbol نشان داده می‌شود مقایسه می‌کند. اگر lookahead با symbol ارسالی به تابع (نماد مورد انتظار تجزیه کننده) برابر

باشد در این صورت lookahead به نماد بعدی رشته ورودی اشاره می‌کند، در غیر این صورت خطا است و پیغام خطا داده می‌شود و تجزیه کننده خاتمه می‌یابد. پیاده‌سازی match() به صورت ذیل است.

```
void   match(char symbol){
if ( lookahead== symbol)
    lookahead=getche();
else{
    cout<<" error";
   exit(0);
   }
return;
}
```

در ذیل به شرح جزیات پیاده سازی توابع term(),rest() و expr() می‌پردازیم.

تابع term(): با توجه به term→1|2|3 ، سه انتخاب term→1 ، term→2 و term→3 وجود دارد که برای انتخاب بین این سه از lookahead استفاده می‌کنیم. تابع term() به صورت ذیل پیاده‌سازی می‌شود.

```
void term(){
                                    اگر lookahead='1' باشد term→1 انتخاب می‌شود//
if (lookahead=='1')
                                    پیشروی lookahead در رشته ورودی//
   match('1');
                                    اگر lookahead='2' باشد term→2 انتخاب می‌شود//
else if (lookahead=='2')
                                    پیشروی lookahead در رشته ورودی//
   match('2');
                                    اگر lookahead='3' باشد term→3 انتخاب می‌شود//
else if (lookahead=='3')
                                    پیشروی lookahead در رشته ورودی//
   match('3');

else {cout<<error";
      exit(0);
      }
}
```

تابع expr(): با توجه به expr→ term rest و با توجه به اینکه برای term و rest توابع term() و rest() ایجاد می‌شود در نتیجه تابع expr() این توابع را به ترتیب فراخوانی می‌کند. تابع expr() به صورت ذیل پیاده سازی می‌گردد.

```
void expr(){
  term();
  rest();
return;
```

}

تابع rest() : با توجه به rest→'+' expr | '-' expr | '*' expr | '/' expr | ∈ ، برای rest انتخابهای مختلفی وجود دارد. انتخاب بین آنها با توجه به lookahead انجام می‌شود. اگر نشانه جاری که در lookahead قرار دارد با نماد شروع هر یک از قواعد تولید برابر باشد، قاعده تولید متناظر آن انتخاب می‌گردد. به عنوان مثال اگر lookahead='+' باشد rest→'+' expr انتخاب می‌شود. اگر lookahead='-' باشد rest→'-' expr انتخاب می‌شود. اگر lookahead='*' باشد rest→'*' expr انتخاب می‌شود و اگر lookahead='/' باشد rest→'/' expr انتخاب می‌شود.

اگر lookahead='\n' باشد رشته ورودی تمام شده و رشته ورودی پذیرفته می‌گردد. اگر lookahead برابر هیچ یک از علائم -,*,+,/ و \n نباشد، rest→∈ انتخاب می‌گردد. تابع rest() به صورت ذیل پیاده سازی می‌شود.

```
void rest(){
                                    اگر '+'=lookahead باشد rest→+ expr انتخاب می‌شود//
if (lookahead=='+'){
    match('+');
    expr();
}
                                    اگر '-'=lookahead باشد rest→- expr انتخاب می‌شود//
else if (lookahead =='-'){
        match('-');
        expr();
}
                                    اگر '*'=lookahead باشد rest→* expr انتخاب می‌شود//
else if (lookahead =='*'){
        match('*');
        expr();
}
                                    اگر '/'=lookahead باشد rest→/ expr انتخاب می‌شود//
else if (lookahead =='/'){
        match('/');
        expr();
}
                                    //انتخاب rest→∈
else;
return;
}
```

غیرپایانه شروع گرامر expr است. در نتیجه فراخوانی توابع از expr() شروع می‌گردد. lookahead در شروع تجزیه، به اولین نماد رشته ورودی اشاره می‌کند. در نتیجه در ابتدا lookahead=getche(); قرار می‌دهیم. این عبارت باعث می‌شود lookahead حاوی اولین کاراکتر یا نماد شروع رشته ورودی باشد. پیاده‌سازی main() به صورت ذیل است.

```
int main(){
lookahead= getche();
expr();
cout<<"accepted";
return 0;
}
```

با توجه به مطالب ذکر شده برنامه کامل تجزیه کننده پیشگوی گرامر به صورت ذیل است.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>


char  lookahead;
void term();
void rest();

void   match(char symbol){
if ( lookahead== symbol)
    lookahead=getche();
else{
    cout<<" error";
   exit(0);
   }
return;
}

void expr(){
term();
rest();
}

void term(){
if (lookahead=='1')
   match('1');
else if (lookahead=='2')
   match('2');
else if (lookahead=='3')
   match('3');
else {cout<<"error";
      exit(0);
}
return ;
}

void rest(){
if (lookahead=='+'){
   match('+');
    expr();
}
else if (lookahead =='-'){
match('-');
expr();
}
```

```
else if (lookahead =='*'){
match('*');
expr();
}
else if (lookahead =='/'){
match('/');
expr();
}
else;
return;
}

int main(){
lookahead= getche();
expr();
cout<<"accepted";
return 0;
}
```

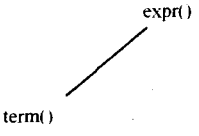
آنگونه که در مباحث ساختمان داده‌ها، درخت را با اشاره‌گرها می‌سازیم، در تجزیه کننده‌ها درخت تجزیه ساخته نمی‌شود، بلکه تجزیه کننده به گونه‌ای رفتار می‌کند که گویی درخت تجزیه ایجاد شده است اگر فراخوانی توابع را به صورت درخت نشان دهیم، درختی مانند درخت تجزیه به دست می‌آید. برای نمونه برای رشته 1+2 باشد مراحل ذیل انجام می‌شود در هر مرحله به روند فراخوانی توابع دقت کنید.

۱- lookahead=1

۲- expr() فراخوانی می‌شود.

شکل ۳-۱۸ فراخوانی expr()

۳- expr() تابع term() را فراخوانی می‌کند. در نتیجه:

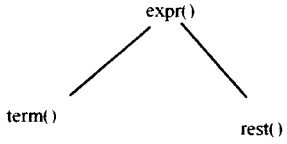


شکل ۳-۱۹ فراخوانی term()

٤- شرط if(lookahead=='1') صحیح است، match('1') اجرا می‌شود و lookahead به نماد بعدی اشاره می‌کند، در نتیجه: lookahead= +

۵- بازگشت به expr()

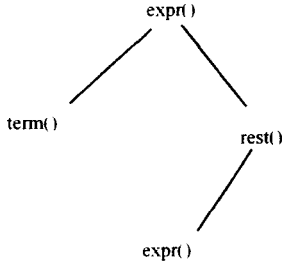
۶- expr() تابع rest() را فراخوانی می‌کند.



شکل ۳-۲۰ فراخوانی rest()

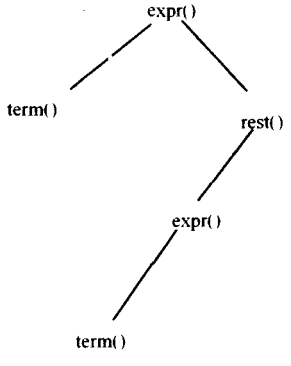
۷- شرط if(lookahead=='+') صحیح است، match('+') اجرا می‌شود و lookahead به نماد بعدی اشاره می‌کند، در نتیجه: lookahead=2

۸- rest() تابع expr() فراخوانی می‌شود.



شکل ۳-۲۱ فراخوانی expr()

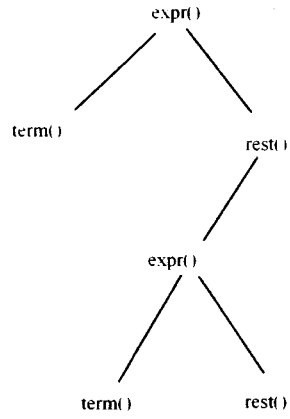
۹- expr() تابع term() را فراخوانی می‌کند.



شکل ۳-۲۲ فراخوانی term()

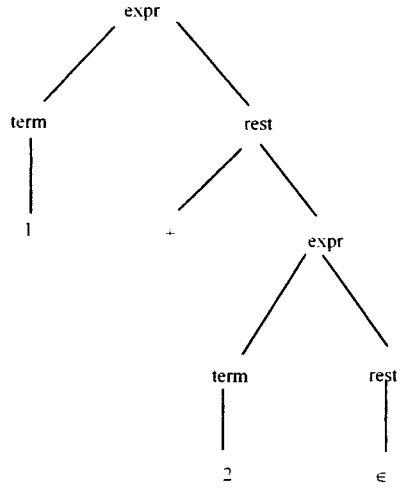
۱۰- شرط if(lookahead=='2') صحیح است، match('2') اجرا می‌شود و lookahead به نماد بعدی اشاره می‌کند، در نتیجه lookahead= \n

۱۱- expr() تابع rest() را فراخوانی می‌کند.



شکل ۳-۲۳ فراخوانی rest()

۱۲- هیچ یک از شرط‌های rest() برقرار نیست، در نتیجه به expr() باز می‌گردد.
۱۳- بازگشت از توابع تا رسیدن به main() اجرا شده و در نهایت accepted چاپ می‌گردد.
اگر درخت تجزیه را برای 1+2 بسازیم شکل ۳-۲٤ به دست می‌آید.



شکل ۳-۲٤ درخت تجزیه 1+2

اگر درخت فراخوانی توابع را با درخت تجزیه مقایسه کنیم، مشخص می‌شود که فراخوانی توابع منطبق بر گره‌های داخلی (غیرپایانه‌های گرامر) درخت تجزیه است.

در ساخت تجزیه کننده پیشگو نکاتی وجود دارد که هر یک از این نکات را به وسیله مثال‌هایی بیان می‌کنیم.

مثال۳-۳۱ فرض کنید می‌خواهیم برای گرامر ذیل تجزیه کننده پیشگو ایجاد کنیم.

$A \rightarrow B \mid C$
$B \rightarrow bB \mid f$
$C \rightarrow cC \mid e$

برای پیاده‌سازی تجزیه کننده این گرامر سه تابع A(),B(),C() مورد نیاز است. با توجه به A→B|C در پیاده سازی تابع مربوط به غیر پایانه A می‌بایست بین دو تابع C(), B() یکی انتخاب و فراخوانی گردد. اما انتخاب بین C() و B() بر چه اساسی باید انجام شود. تنها راه حل ممکن، مقایسه نماد جاری ورودی(lookahead) با مجموعه نمادهایی است که رشته‌های مشتق شده از هر یک از غیرپایانه‌های B,C با آنها شروع می‌شوند.

- اگر نماد lookahead در مجموعه نمادهایی باشد که رشته‌های مشتق شده از B با آنها شروع می‌شوند، آنگاه B() انتخاب می‌گردد. بنابراین اگر lookahead=b و lookahead=f باشد، B() انتخاب می‌شود زیرا فقط رشته‌های مشتق شده از B می‌توانند با b و یا f شروع شوند. به عبارت دیگر اگر lookahead در first(B)={b,f} باشد، B() فراخوانی می‌شود.

- اگر نماد lookahead در مجموعه نمادهایی باشد که رشته‌های مشتق شده از C با آنها شروع می‌شود، آنگاه C() انتخاب می‌گردد. بنابراین اگر lookahead=e و lookahead=c باشد، C() انتخاب می‌شود زیرا فقط رشته‌های مشتق شده از C می‌توانند با e و یا c شروع شوند. به عبارت دیگر اگر lookahead در first(C)={c,e} باشد، C() فراخوانی می‌شود.

با توجه به A→B|C و first(C)={c,e} و first(B)={b,f} تابع A() را به صورت ذیل پیاده‌سازی می‌کنیم.

```
void A(){
        اگر lookahead در first(B)={b,f} باشد، A→B انتخاب می‌شود.//
if ( lookahead=='b' || lookahead=='f')
    B();
          اگر lookahead در first(C)={c,e} باشد، A→C انتخاب می‌شود.//
else if ( lookahead=='c' || lookahead=='e')
    C();
}
else { cout<<"error";
      exit(0);
}
return;
}
```

برای پیاده سازی B() نیز باید بین B→bB و B→f یکی را انتخاب شود. برای انتخاب، از مقایسه lookahead با مجموعه first(bB)={b} و first(f)={f} استفاده می‌کنیم. در نتیجه:

```
void B(){
                    اگر lookahead در {first(bB)={b باشد B→bB انتخاب می‌گردد.//
if (lookahead=='b'){
   match('b');
   B();
}
                    اگر lookahead در {first(f)={f باشد B→f انتخاب می‌گردد.//
else if (lookahead=='f')
      match('f');
else {
     cout<<"error";
     exit(0);
    }
}
```

برای پیاده سازی C() نیز باید بین C→cC و C→e یکی را انتخاب شود. برای انتخاب، از مقایسه lookahead با مجموعه {first(cC)={c و {first(e)={e استفاده می‌کنیم. در نتیجه:

```
void C(){
                    اگر lookahead در {first(cC)={c باشد C→cC انتخاب می‌گردد.//
if (lookahead=='c'){
   match('c');
   C();
}
                    اگر lookahead در {first(e)={e باشد C→e انتخاب می‌گردد.//
else if (lookahead=='e')
      match('e');
else {
     cout<<"error";
     exit(0);
    }
}
```

سوالی که مطرح می‌شود این است که آیا برای هر گرامری می‌توان یک تجزیه کننده پیشگو ایجاد کرد برای پاسخ به این سوال به گرامر ذیل توجه کنید:

exp→ expr + term | term
term → 1 | 2 | 3

برای این گرامر به روشی که ذکر گردید یک تجزیه کننده پیشگو به صورت ذیل ایجاد می‌کنیم.

```
char   lookahead;
expr(){
        expr();
        match('+');
        term();
       }

void term(){
if (lookahead=='1')
   match('1');
```

```
else if (lookahead=='2')
   match('2');
else if (lookahead=='3')
   match('3');
return 0;
}

int main(){
   lookahead=getche();
   expr();
   return 0;
}
```

حال با توجه به رشته 2+1 اجرای این برنامه را بررسی می‌کنیم. اجرای برنامه با فراخوانی expr() آغاز می‌گردد.

```
expr(){
        expr();
        match('+');
        term();
        }
```

تابع expr() در اولین دستور ;expr() را فراخوانی می‌کند. این فراخوانی منجر به فراخوانی مجدد ;expr() تا بینهایت خواهد شد. در نتیجه برنامه هیچ وقت از حلقه خارج نمی‌شود.

گرامر فوق نشان می‌دهد که برای هر گرامری نمی‌توان تجزیه کننده پیشگو ایجاد کرد. با بررسی گرامر مشخص می‌شود که علت بروز این مشکل بازگشتی چپ است. در واقع برای گرامرهایی که دارای بازگشتی چپ هستند نمی‌توان تجزیه کننده پیشگو ایجاد کرد مگر آنکه بازگشتی چپ به روشی که قبلا ذکر شد حذف شود.

مثال۳-۳۲ می‌خواهیم برای گرامر ذیل، تجزیه کننده پیشگو بسازیم.

$A \rightarrow a\ A | a\ B$
$B \rightarrow b\ B | c$

با توجه به $A \rightarrow aA | aB$ تابع A() باید بین aA و aB یکی را انتخاب کند. با توجه به تعریف first:

$first(aA) = \{a\}$
$first(aB) = \{a\}$

با توجه به اینکه a در first(aB) و first(aA) قرار دارد، بنابراین، aA و aB با a شروع می‌شوند. به همین دلیل اگر نماد جاری a باشد تابع A() نمی‌تواند بین aA و aB یکی را انتخاب کند زیرا هر دو با a شروع می‌شوند. این مشکل برخورد first/first نامیده می‌شود. به عنوان یک قانون کلی اگر قاعده تولید A را به صورت ذیل در نظر بگیریم.

$A \rightarrow \alpha_1 | \alpha_2 | \alpha_3 | ... | \alpha_n$

اگر رابطه ذیل حداقل بین دو α_i و α_j به طوریکه $i \neq j$ برقرار باشد برخورد first / first رخ می‌دهد.

$first(\alpha_i) \cap first(\alpha_j) \neq \varnothing$

این رابطه بیان می‌کند که اگر حداقل دو انتخاب α_i و α_j با یک نماد شروع شوند برخورد first/first رخ می‌دهد. برای رفع برخورد first/first می‌توان از فاکتورگیری چپ که قبلا بیان شد استفاده کرد.

مثال ۳-۳۳ با استفاده از فاکتورگیری چپ برخورد first/first گرامر مثال ۳-۳۲ را حذف کرده سپس تجزیه کننده آن را ایجاد کنید.

پس از فاکتور گیری چپ گرامر به صورت ذیل تبدیل می‌گردد.

```
A→ aR
R→ A|B
B→ bB|c
```

پس ازحذف برخورد first/first، برای گرامر جدید به دست آمده می‌توان تجزیه کننده پیشگو ایجاد کرد. در پیاده سازی R بین A و B یکی انتخاب می‌شود، با توجه به آنچه در مورد first ذکر گردید ابتدا first(A) و first(B) را محاسبه می‌کنیم.

```
first(A)={a}
first(B)={b,c}
```

تجزیه کننده به صورت ذیل پیاده سازی می‌گردد.

```
void A(){
   match('a');
   R();
}

void R(){
                        اگر lookahead در {a}=first(A) باشد، R→A انتخاب می‌شود.//
if(lookahead=='a')
      A();
                        اگر lookahead در {b,c}=first(B) باشد، R→B انتخاب می‌شود. //
else if (lookahead=='b' || lookahead=='c')
     B();
    اگر lookahead در {b,c}=first(B) و {a}=first(A) نباشدخطا است و رشته رد می‌شود. //
else{ cout<"error";
      exit(0);
    }
}
void B(){
                        اگر lookahead در {b}=first(bB) باشد، B→ bB انتخاب می‌شود.//
if(lookahead=='b'){
    match('b');
    B();
}
                        اگر lookahead در {c}=first(c) باشد، B→c انتخاب می‌شود.//
else if(lookahead=='c'){
match('c');
cout<<"Accepted";
```

```
exit(0);
  }
}

int main(){
   lookahead= getche();
   A();
   return 0;
}
```

مثال۳-۳٤ برای گرامر ذیل تجزیه کننده پیشگو ایجاد کنید.

$A \rightarrow Bed$

$B \rightarrow e|a| \in$

با توجه به مطالب ذکر شده تجزیه کننده این گرامر را به صورت ذیل پیاده‌سازی می‌کنیم.

```
char  lookahead;
void A(){
    B();
    match('e');
    match('d');
}

void B(){
if (lookahead =='e'){
      match('e');
else  if (lookahead =='a')
      match('a');
else ;
return ;
}
int main(){
    lookahead=getche();
    A();
    return 0;
}
```

گرامر مورد نظر سه رشته eed,aed وed را تولید می‌کند. رفتار تجزیه کننده را در مقابل دریافت ed و eed بررسی می‌کنیم. اگر رشته ورودی eed باشد مراحل ذیل اجرا می‌شود.

۱- main() اجرا می‌گردد.

۲- lookahead='e'

۳- روال A()، روال B() را فراخوانی می‌کند.

٤- نتیجه مقایسه if (lookahead=='e') صحیح است، در نتیجه match('e'); اجرا شده و lookahead به نماد بعدی اشاره می‌کند، با توجه به ورودی lookahead= 'e' می‌گردد.

۵- به روال A(); بر می‌گردد.

٦- match('e'); اجرا می‌شود چون lookahead='e' است در نتیجه lookahead به نماد بعدی اشاره می‌کند بنابراین lookahead='d'.

۷- match('d'); اجرا می‌شود چون lookahead='d' است در نتیجه lookahead به نماد بعدی اشاره می‌کند.

۸- در نتیجه پیغام Accepted چاپ شده و ورودی پذیرفته می‌گردد.

اگر رشته ورودی ed باشد مراحل ذیل اجرا می‌شود.

۱- main() اجرا می‌گردد.

۲- lookahead='e'

۳- روال A()، روال B() را فراخوانی می‌کند.

٤- نتیجه مقایسه if (lookahead=='e') صحیح است، در نتیجه match('e'); اجرا شده و lookahead به نماد بعدی اشاره می‌کند، با توجه به ورودی lookahead='d' می‌شود.

۳- به روال A(); بر می‌گردد.

٦- نتیجه مقایسه match('e'); صحیح نیست.

۷- cout<<"error: " اجرا می‌گردد و علی رغم اینکه ed توسط گرامر قابل تولید است ولی رشته ed رد می‌شود.

نماد e مسبب بروز مشکل است، زیرا اگر نماد ورودی e باشد دو حالت ذیل محتمل است:

۱- e از B→e تولید شده است.

۲- e از A→Bed به دست آمده است و B تهی در نظر گرفته شده است.

با توجه به احتمالات ذکر شده اگر نماد ورودی e باشد، تجزیه کننده نمی‌تواند بین حالات بالا یکی را انتخاب کند. علت بروز مشکل آن است که نمادی که B با آن شروع می‌شود با نمادی که بعد از B می‌آید یکسان است و B می تواند تهی نیز شود.

با توجه به تعریف follow ، مشکل زمانی بروز می‌کند که در قاعده تولیدی به A→α|β شرایط ذیل برقرار باشد.

۱- β بتواند ∈ را تولید کند.

۲- حداقل یک نماد وجود دارد که هم می‌تواند در شروع α و هم بعد از A باشد. یا به عبارت دیگر رابطه ذیل برقرار باشد.

$first(\alpha) \cap follow(A) \neq \varnothing$

در این شرایط برخورد first/follow رخ می‌دهد.

مثال۳-۳۵ در گرامر ذیل e مسبب برخورد first/follow است.

A→ Bed
B→ e|a|∈

اگر $\alpha=e$ و $\beta=\in$ باشد،

$first(e)=\{e\}$
$follow(B)=\{e\}$

در نتیجه:

$first(e) \cap follow(B)=\{e\}\neq\varnothing$

در چنین گرامرهایی که برخورد first/follow رخ می‌دهد، ابتدا باید با تغییر گرامر مانند جایگزینی غیر پایانه‌ها با سمت راست قاعده تولید آن، برخورد را از بین برد تا بتوان برای گرامر مورد نظر تجزیه کننده پیشگو ایجاد کرد. اگر A دارای n انتخاب مختلف باشد، سمت راست α نیز n بار تکرار شده و در هر تکرار به جای A یکی از انتخابها قرار می‌گیرد. به عنوان مثال گرامر قبل که دارای برخورد first/follow بود را می‌توان به صورت ذیل تبدیل کرد، که برخورد first/follow نیست.

$A\rightarrow$ eed | aed | ed

اما انجام جایگزینی ممکن است باعث بروز برخورد first/first شود. به عنوان مثال $A\rightarrow$eed|ed دارای برخورد first/first است برای رفع برخورد first/first را نیز می‌توان با فاکتور گیری چپ رفع کرد.

$A\rightarrow$ eR|aed
$R\rightarrow$ed|d

البته گرامرهایی وجود دارند که با هر نوع تغییر در گرامر نمی توان تجزیه کننده پیشگو برای آن ایجاد کرد.

از مزایای تجزیه کننده پیشگو این است که می‌توان این نوع تجزیه کننده را به سادگی برای گرامرها تولید کرد. ولی این نوع تجزیه کننده به صورت توابع بازگشتی پیاده سازی می‌شود در نتیجه سرعت اجرای آن کند است.

نکات ذکر شده در مورد ساخت تجزیه کننده پیشگو را می‌توان به صورت ذیل خلاصه کرد.

- تجزیه کننده پیشگو برای هرقاعده تولید $A\rightarrow\alpha_1|\alpha_2...|\alpha_n$ یک تابع ایجاد می‌کند.
- اگر نماد جاری رشته ورودی (lookahead) فقط در مجموعه $first(\alpha_i)$ باشد، α_i انتخاب می‌گردد.
- اگر نماد جاری رشته ورودی در هیچ یک از مجموعه‌های $first(\alpha_i)$ نباشد و غیرپایانه A بتواند $\in$ را تولید کند، در این صورت $\in$ انتخاب می‌گردد.

در حالات ذیل نمی‌توان تجزیه کننده را ایجاد کرد.

- اگر در $A\rightarrow\alpha|\beta$ نمادی در مجموعه‌های first(β) و first(α) مشترک باشد، برخورد first/first رخ می‌دهد و در نتیجه نمی‌توان تجزیه کننده پیشگو برای گرامر ایجاد کرد.
- اگر در $A\rightarrow\alpha|\beta$ ، نمادی در مجموعه‌های first(α) و follow(A) مشترک باشد و β بتواند $\in$ را تولید کند، در این صورت برخورد first/follow رخ می‌دهد و در نتیجه نمی‌توان تجزیه کننده پیشگو ایجاد کرد.

۳-۱٤ تجزیه کننده پیشگوی غیر بازگشتی

در بخش قبل روش ساخت یک تجزیه کننده بالا به پایین به صورت بازگشتی ارائه گردید. در تجزیه کننده پیشگو برای انتخاب قاعده تولید مناسب در $A\rightarrow\alpha_1|\alpha_2...|\alpha_n$ ، نماد جاری (lookahead) با $first(\alpha_1)$، $first(\alpha_2)$ ... و $first(\alpha_n)$ با استفاده از دستور if مقایسه می‌گردید. به عبارت دیگر در زمان اجرا مشخص می‌شد با توجه به نماد جاری ورودی، α_i مناسب کدام است و این کار مستلزم اجرای چندین دستور if در برنامه است که سرعت اجرای برنامه را کند می‌کند. یکی از روشهای حل این مشکل استفاده از جدول تجزیه است در این روش برای هر غیر پایانه، متناسب با هر پایانه ورودی، قاعده تولید مناسب را در جدولی ذخیره می‌کنیم و تجزیه کننده از این جدول برای انتخاب قاعده تولید مناسب استفاده می‌کند. گرامر ذیل که برای آن تجزیه کننده پیشگو تولید گردید را در نظر می‌گیریم.

$A \rightarrow B \mid C$
$B \rightarrow bB \mid f$
$C \rightarrow cC \mid e$

در بخش قبل مشخص شد که:

اگر در A() نماد جاری b و یا f باشد $A\rightarrow B$ انتخاب می‌شود و اگر نماد جاری c و یا e باشد $A\rightarrow C$ انتخاب می‌شود. این انتخابها را می‌توان در سطر اول جدولی مانند جدول ۳-۲ ذخیره کرد.

در مورد B نیز بیان شد که اگر نماد جاری b باشد $B\rightarrow bB$ و اگر نماد جاری f باشد $B\rightarrow f$ انتخاب می‌شود. این انتخابها را می‌توان در سطر دوم جدولی مانند جدول ۳-۲ ذخیره کرد.

در مورد C نیز بیان شد که اگر نماد جاری c باشد $C\rightarrow cC$ و اگر نماد جاری e باشد $C\rightarrow e$ انتخاب می‌شود. این انتخابها را می‌توان در سطر سوم جدولی مانند جدول ۳-۲ ذخیره کرد.

موارد ذکر شده را می‌توان به صورت زیر ذخیره کرد.

جدول ۳-۲ جدول تجزیه

	b	f	c	e
A	A → B	A → B	A→C	A→C
B	B→ bB	B→f		
C			C→cC	C→e

این جدول می‌تواند در روند تجزیه مورد استفاده تجزیه کننده قرار گیرد به عنوان مثال اگر غیرپایانه‌ای که باید گسترش داده شود A و نماد جاری ورودی e باشد با توجه به جدول (M[A,e]=A→C) قاعده تولید A→C انتخاب می‌گردد و نیاز به اجرای چندین دستور if نخواهد بود.

همچنین اگر غیرپایانه جاری B و نماد جاری ورودی e باشد با توجه به جدول تجزیه M[B,e] خالی است در نتیجه خطا رخ داده و رشته ورودی رد می شود.

ایده‌ای که در مورد استفاده از جدول به جای اجرای دستورات if ذکر گردید در تجزیه کننده پیشگوی غیربازگشتی به کار می‌رود. همچنین هر برنامه بازگشتی را می‌توان به صورت غیر بازگشتی نیز پیاده‌سازی کرد، برای تبدیل برنامه بازگشتی به غیر بازگشتی از پشته استفاده می‌کنیم. در این بخش نحوه ساخت تجزیه کننده پیشگوی غیربازگشتی[1] که یک تجزیه کننده بالا به پایین است را مورد بررسی قرار می‌دهیم. شکل ۳-۲۵ قسمتهای مختلف تجزیه کننده پیشگوی غیربازگشتی را نشان می‌دهد.

قسمتهای مختلف تجزیه کننده پیشگوی غیر بازگشتی عبارتند از:

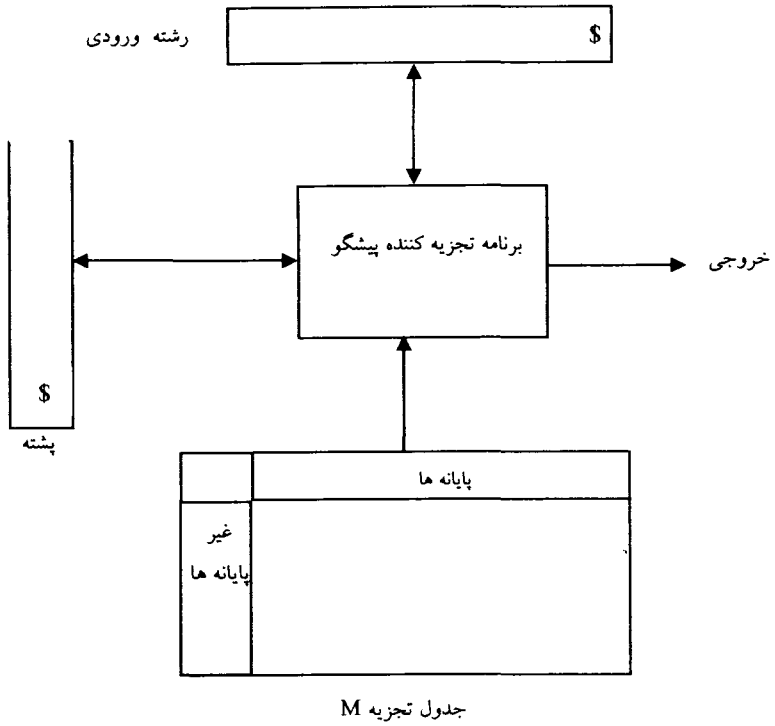
۱- رشته ورودی: این مکان، محلی برای نگهداری رشته ورودی است. به منظور نشان دادن پایان رشته، علامت $ به پایان رشته ورودی اضافه می‌گردد.

۲- پشته: محلی برای نگهداری نمادهای گرامر است. این پشته در ابتدا شامل علامت $ و نماد شروع گرامر در بالای پشته است.

۳- جدول تجزیه: این جدول با توجه به نشانه جاری از رشته ورودی و غیرپایانه بالای پشته قاعده تولید بعدی که باید مورد استفاده قرار گیرد، را مشخص می‌کند. برای هر گرامری جدول تجزیه جداگانه‌ای ایجاد می‌شود. اگر a نشان دهنده نشانه جاری رشته ورودی و A غیرپایانه بالای پشته (غیر پایانه‌ای که باید گسترش داده شود) و M جدول تجزیه باشد،

---

۱. تجزیه کننده پیشگوی غیر بازگشتی، تجزیه کننده LL(1) نیز نامیده می‌شود.



شکل ۳-۲۵ قسمتهای مختلف تجزیه کننده پیشگوی غیر بازگشتی

آنگاه M[A,a] قاعده تولید بعدی را برای گسترش A مشخص می‌کند (مانند آنچه در مورد استفاده از جدول در تجزیه پیشگو ذکر شد). نحوه ایجاد این جدول در بخشهای بعدی تشریح خواهد شد.

٤- برنامه تجزیه کننده پیشگو: این بخش نشان دهنده برنامه‌ای است که با توجه به نماد ورودی و نماد بالای پشته و جدول تجزیه، پشته را بروز کرده و در صورت نیاز نشانگر رشته ورودی را به جلو حرکت می‌دهد.

٥- خروجی: دنباله ای از قواعد تولید گرامر است که در تجزیه استفاده شده‌اند.

این تجزیه کننده دارای دو انتقال به نام انتقال پیشگو و انتقال تطبیق است. نماد بالای پشته و نشانه جاری ورودی مشخص می‌کند کدام یک از این دو انتقال باید اجرا شود. این دو انتقال به صورت ذیل تعریف می‌شوند.

۱- انتقال پیشگو: اگر بالای پشته غیر پایانه X و نماد جاری رشته ورودی a باشد، X از پشته حذف شده و قاعده تولید M[X,a] از جدول استخراج شده و به صورت معکوس در پشته

درج می‌شود. به عنوان مثال اگر M[X,a]=X→αβγ باشد، α در بالای پشته قرار می‌گیرد. در واقع با این عمل مانند تجزیه پیشگو، غیرپایانه گسترش می‌یابد. اگر M[X,a] خالی باشد خطا رخ داده است.

۲- انتقال تطبیق: اگر نماد بالای پشته، پایانه باشد و این پایانه با نشانه جاری رشته ورودی یکسان باشد، آنگاه پایانه بالای پشته حذف می‌گردد و نشانه بعدی در رشته ورودی به عنوان نشانه جاری در نظر گرفته می‌شود، ولی اگر نشانه بالای پشته و نشانه جاری رشته ورودی یکسان نباشد خطا رخ داده است.

تجزیه کننده دو انتقال تطبیق و پیشگو را دائما انجام می‌دهد. تجزیه وقتی تمام می‌شود که پشته و رشته ورودی خالی شده باشد.

۳-۱۵ ساخت جدول تجزیه پیشگوی غیر بازگشتی

با توجه به آنچه در مورد استفاده از جدول ذکر شده در این بخش نحوه ساخت جدول تجزیه برای تجزیه کننده پیشگوی غیر بازگشتی را تشریح می‌کنیم. با توجه به مجموعه‌های first و follow و با استفاده از گرامر ذیل مراحل ساخت جدول تجزیه را شرح می‌دهیم.

expr → term rest
rest → + expr | - expr | ∈
term → id

ابتدا first تمام سمت راستهای قواعد تولید و follow تمام غیر پایانه‌ها را محاسبه می‌کنیم.

first(term rest)=first(term)={id}
first(+expr)=first(+)={+}
first(-expr)=first(-)={-}
first(∈)={∈}
first(id)={id}
follow(expr)={$}
follow(term)={+,-,$}
follow(rest)={$}

جدول تجزیه M به صورت ذیل است.

جدول ۳-۳ ساختار جدول تجزیه پیشگوی غیر بازگشتی

	id	+	-	$
expr				
term				
rest				

در ستونهای جدول تجزیه، پایانه‌های گرامر و $ (نشان دهنده پایان رشته ورودی)، و در سطرها غیرپایانه‌های گرامر قرار دارد. M[X,a] مشخص می‌کند اگر نماد جاری ورودی a و غیرپایانه بالای پشته X باشد، X را چگونه باید گسترش داد. با استفاده از first و follow که در مراحل قبل محاسبه شده‌اند جدول تجزیه را مرحله به مرحله کامل می‌کنیم.

- قاعده تولید expr→term rest را در نظر می‌گیریم. گسترش expr به term rest زمانی امکان پذیر است که نماد جاری ورودی، نمادی باشد که رشته مشتق شده از دنباله term rest بتواند با آن شروع شود، به عبارت دیگر نماد جاری ورودی در first(term rest) باشد.

با توجه به first(term rest)=first(term)={id} ، گسترش expr به term rest وقتی امکان‌پذیر است که نماد جاری ورودی id باشد. در نتیجه:

M[expr,id]= expr→term rest

این وارده از جدول M نشان می‌دهد، هرگاه غیر پایانه جاری (غیر پایانه بالای پشته) expr و نماد ورودی id باشد، expr را به term rest گسترش می‌دهیم. برای expr انتخاب دیگری وجود ندارد، در نتیجه غیرپایانه بعدی را بررسی می‌کنیم.

۲- قاعده تولید rest → +expr | -expr | ∈ را در نظر می‌گیریم. rest شامل سه انتخاب است هر یک را جداگانه بررسی می‌نیم.

- گسترش rest به expr+ زمانی امکان پذیر است که نماد جاری ورودی، نمادی باشد که رشته مشتق شده از دنباله expr+ بتواند با آن شروع شود، به عبارت دیگر نماد جاری ورودی در first(+expr) باشد. با توجه به first(+expr)=first(+)={+} ، گسترش rest به expr+ وقتی امکان‌پذیر است که نماد جاری ورودی + باشد. در نتیجه:

M[rest,+]= rest→+expr

این وارده از جدول M نشان می‌دهد، هرگاه غیرپایانه جاری (غیرپایانه بالای پشته) rest و نماد ورودی + باشد، rest را به expr+ گسترش می‌دهیم.

- گسترش rest به expr- زمانی امکان پذیر است که نماد جاری ورودی، نمادی باشد که رشته مشتق شده از دنباله expr- بتواند با آن شروع شود، به عبارت دیگر نماد جاری ورودی در first(-expr) باشد. با توجه به first(-expr)=first(-)={-} ، گسترش rest به expr- وقتی امکان پذیر است که نماد جاری ورودی - باشد. در نتیجه:

M[rest,-]= rest→-expr

این وارده از جدول M نشان می‌دهد، هرگاه غیرپایانه جاری (غیرپایانه بالای پشته) rest و نماد ورودی - باشد، rest را به expr- گسترش می‌دهیم.

-گسترش rest به ∈ نیاز به بررسی بیشتری دارد، اگر نماد جاری ورودی - یا + باشد به ترتیب expr- و expr+ انتخاب می‌گردد. اگر نماد ورودی + و - نباشد، می‌توان rest→∈ را انتخاب کرد (مانند تجزیه پیشگو) ولی این روش را می‌توان هوشمندانه‌تر کرد. اگر نماد جاری ورودی بتواند بعد از rest قرار گیرد در این صورت rest→∈ مجاز است. نمادهایی که می‌توانند بعد از rest قرار گیرند با follow(rest) محاسبه می‌شود. در نتیجه rest→∈ برای پایانه‌های follow(rest)={$} صحیح است. در نتیجه:

M[rest,$]= rest→∈

برای درک بهتر این نکته فرض کنید نماد ورودی a باشد به طوریکه a در follow(B) نباشد، در این شرایط اگر B→∈ مورد استفاده قرار گیرد، دنباله Ba به دست می‌آید، که a بلافاصله بعد از B است یعنی a در follow(B) است که با فرض اول در تناقض است.

- قاعده تولید term→id را در نظر می‌گیریم.

گسترش term به id زمانی امکان پذیر است که نماد جاری ورودی، نمادی باشد که id بتواند با آن شروع شود، به عبارت دیگر نماد جاری ورودی در first(id) باشد. با توجه به first(id)={id} ، گسترش term به id وقتی امکان پذیر است که نماد جاری ورودی id باشد. در نتیجه:

M[term,id]= term→id

این وارده از جدول M نشان می‌دهد، هرگاه غیر پایانه جاری (غیر پایانه بالای پشته) term و نماد ورودی id باشد، term را به id گسترش می‌دهیم. نتیجه انجام این مرحله در جدول ۳-٤ نشان داده شده است. خانه‌های خالی با error پر می‌شود و یا برای اختصار خالی باقی می‌ماند. اگر تجزیه کننده به این حالات برسد خطا رخ داده است.

جدول ۳-٤ جدول تجزیه پیشگوی غیر بازگشتی

	id	+	-	$
expr	expr→term rest			
term	term→id			
rest		rest→+expr	rest→ - expr	rest→∈

مراحل ساخت جدول تجزیه را می‌توان برای هر قاعده تولید A→α در سه مرحله ذیل خلاصه کرد.

۱- برای هر پایانه a (به جز ∈) در مجموعه first(α) قانون A→α به جدول در M[A,a] اضافه می‌گردد.

۲- اگر ∈ در first(α) باشد برای هر پایانه b در follow(A)، M[A,b]=A→∈ می‌گردد.
۳- در خانه‌های خالی جدول، error قرار می‌دهیم.
نحوه عملکرد تجزیه کننده را با توجه به رشته id+id-id و جدول تجزیه به دست آمده مورد بررسی قرار می‌دهیم. مراحل تجزیه در جدول ۳-۵ نشان داده شده است.

جدول ۳-۵ مراحل تجزیه id+id-id

رشته ورودی	پشته	خروجی	توضیحات
id+id-id$	$		به انتهای رشته ورودی و بالای پشته $ درج می‌شود.
id+id-id$	$expr	Expr→term rest	نماد شروع در بالای پشته قرار می‌گیرد. با توجه به اینکه نماد بالای پشته غیر پایانه expr است. یک انتقال پیشگو انجام می‌شود. نشانه جاری در رشته ورودی id و نماد بالای پشته expr است و M[expr,id]=expr→term rest است. در نتیجه expr از بالای پشته حذف و term rest به صورت معکوس وارد پشته می‌شود. نتیجه در مرحله بعد نشان داده شده است.
id+id-id$	$rest term	term→id	با توجه به اینکه نماد بالای پشته غیر پایانه term است. یک انتقال پیشگو انجام می‌شود. نشانه جاری در رشته ورودی id و نماد بالای پشته term است و M[term,id]=term→id است. در نتیجه term از بالای پشته حذف و id به صورت معکوس وارد پشته می‌شود. نتیجه در مرحله بعد نشان داده شده است.
id+id-id$	$rest id		با توجه به اینکه نماد بالای پشته پایانه id است. یک انتقال تطبیق انجام می‌شود. نشانه جاری در رشته ورودی id با نماد بالای پشته id یکسان است. id از بالای پشته حذف شده و نشانگر به جلو حرکت می‌کند در نتیجه نشانه جاری ورودی نشانه + می‌شود. نتیجه در مرحله بعد نشان داده شده است.
+id-id$	$rest	rest→+expr	با توجه به اینکه نماد بالای پشته غیر پایانه rest است. یک انتقال پیشگو انجام می‌شود. نشانه جاری در رشته ورودی + و نماد بالای پشته rest است. با توجه به جدول تجزیه M[rest,+]=rest→+expr است. در نتیجه rest از بالای پشته حذف و +expr به صورت معکوس وارد پشته می‌شود. نتیجه در مرحله بعد نشان داده شده است.
+id-id$	$expr +		با توجه به اینکه نماد بالای پشته پایانه + است. یک انتقال تطبیق انجام می‌شود. نشانه جاری در رشته ورودی + و نماد بالای پشته + است. + از بالای پشته

			حذف شده و نشانگر به جلو حرکت می‌کند در نتیجه نشانه جاری ورودی نشانه id می‌شود. نتیجه در مرحله بعد نشان داده شده است.
id-id$	$expr	expr→term rest	با توجه به اینکه نماد بالای پشته غیر پایانه expr است. یک انتقال پیشگو انجام می‌شود. نشانه جاری در رشته ورودی id و نماد بالای پشته expr است. با توجه به جدول تجزیه M[expr,id]=expr→term rest است. در نتیجه expr از بالای پشته حذف و term rest به صورت معکوس وارد پشته می‌شود. نتیجه در مرحله بعد نشان داده شده است.
id-id$	$rest term	term→id	با توجه به اینکه نماد بالای پشته غیر پایانه term است. یک انتقال پیشگو انجام می‌شود. نشانه جاری در رشته ورودی id و نماد بالای پشته term است. با توجه به جدول تجزیه M[term,id]=term→id است. در نتیجه term از بالای پشته حذف و id به صورت معکوس وارد پشته می‌شود. نتیجه در مرحله بعد نشان داده شده است.
id-id$	$rest id		با توجه به اینکه نماد بالای پشته پایانه id است. یک انتقال تطبیق انجام می‌شود. نشانه جاری در رشته ورودی id و نماد بالای پشته id است. id از بالای پشته حذف شده و نشانگر به جلو حرکت می‌کند در نتیجه نشانه جاری ورودی نشانه - می‌شود. نتیجه در مرحله بعد نشان داده شده است.
-id$	$rest	rest→-expr	با توجه به اینکه نماد بالای پشته غیر پایانه rest است. یک انتقال پیشگو انجام می‌شود. نشانه جاری در رشته ورودی - و نماد بالای پشته rest است. با توجه به جدول تجزیه M[rest,-]=rest→-expr است. در نتیجه rest از بالای پشته حذف و expr - به صورت معکوس وارد پشته می‌شود. نتیجه در مرحله بعد نشان داده شده است.
-id$	$expr -		با توجه به اینکه نماد بالای پشته پایانه - است. یک انتقال تطبیق انجام می‌شود. نشانه جاری در رشته ورودی - و نماد بالای پشته - یکسان است. - از بالای پشته حذف شده و نشانگر به جلو حرکت می‌کند در نتیجه نشانه جاری ورودی نشانهid می‌شود. نتیجه در مرحله بعد نشان داده شده است. نتیجه در مرحله بعد نشان داده شده است.
id$	$expr	expr→term rest	با توجه به اینکه نماد بالای پشته غیر پایانه expr است. یک انتقال پیشگو انجام می‌شود. نشانه جاری در رشته ورودی id و نماد بالای پشته expr است. با توجه به جدول تجزیه M[expr,id]=expr→term rest است. در نتیجه expr از بالای پشته حذف و term

			rest به صورت معکوس وارد پشته می‌شود. نتیجه در مرحله بعد نشان داده شده است.
id$	$rest term	term→id	با توجه به اینکه نماد بالای پشته غیر پایانه term است. یک انتقال پیشگو انجام می‌شود. نشانه جاری در رشته ورودی id و نماد بالای پشته term است. با توجه به جدول تجزیه M[term,id]=term→id است. در نتیجه term از بالای پشته حذف و id به صورت معکوس وارد پشته می‌شود. نتیجه در مرحله بعد نشان داده شده است.
id$	$rest id		با توجه به اینکه نماد بالای پشته پایانه id است. یک انتقال تطبیق انجام می‌شود. نشانه جاری در رشته ورودی id و نماد بالای پشته id است. id از بالای پشته حذف شده و نشانگر به جلو حرکت می‌کند در نتیجه نشانه جاری ورودی نشانه$ می‌شود. نتیجه در مرحله بعد نشان داده شده است.
$	$rest	rest→∈	با توجه به اینکه نماد بالای پشته غیر پایانه rest است. یک انتقال پیشگو انجام می‌شود. نشانه جاری در رشته ورودی$ و نماد بالای پشته rest است. با توجه به جدول تجزیه M[rest,$]=rest→∈ است. در نتیجه rest از بالای پشته حذف و∈ به صورت معکوس وارد پشته می‌شود. نتیجه در مرحله بعد نشان داده شده است.
$	$		با توجه به اینکه نماد بالای پشته پایانه $ است. یک انتقال تطبیق انجام می‌شود. نشانه جاری در رشته ورودی $ و نماد بالای پشته $ یکسان است. $ از بالای پشته حذف شده و نشانگر به جلو حرکت می‌کند. در این وضعیت رشته ورودی تمام شده و پشته نیز خالی شده است.

۳-۱۵-۱ گرامرهای LL(1)

اگر در ساخت درخت تجزیه فقط با رویت یک نشانه بعدی از رشته ورودی در پیمایش چپ به راست، بتوان غیر پایانه بعدی را برای گسترش تشخیص داد در این صورت گرامر مستقل از متن را LL(1) می‌نامیم. دلیل این نامگذاری این است که رشته ورودی از چپ به راست پویش می‌شود[1] و سمت چپ ترین اشتقاق[2] ایجاد می‌شود و عدد یک نیز نشان می‌دهد که فقط دیدن یک نشانه بعدی برای انتخاب قانون تولید بعدی کافی است. LL(1) یکی از ویژگیهای مهم گرامرها است، زیرا ساخت تجزیه کننده را بسیار ساده می‌کند، چون

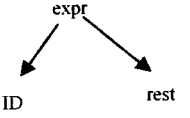
1. Left
2. Leftmost derivation

تجزیه کننده با استفاده از یک نماد جاری می‌تواند به طور دقیق قاعده تولید بعدی را به درستی تعیین کند.

مثال۳-۳۶ گرامر ذیل یک گرامر LL(1) است.

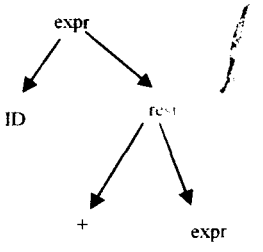
```
expr→ ID rest
rest→'+' expr | '-' expr | ∈
```

به عنوان مثال رشته ID+ID را در نظر می‌گیریم. اولین نماد ID است. واضح است که در این شرایط تنها انتخاب قاعده تولید مناسب expr→ID rest است. درخت تجزیه به صورت شکل ۳-۲۶ تبدیل می‌گردد.



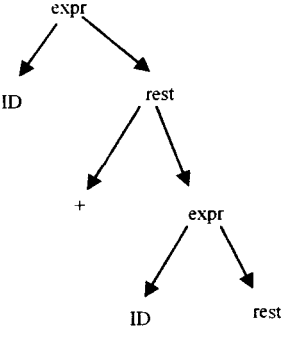
شکل ۳-۲۶ گسترش expr

سه انتخاب برای گسترش rest وجود دارد. فقط با توجه به یک نماد بعدی یعنی + می‌توان تشخیص داد که گسترش rest→'+' expr صحیح است. درخت تجزیه ذیل به دست می‌آید.



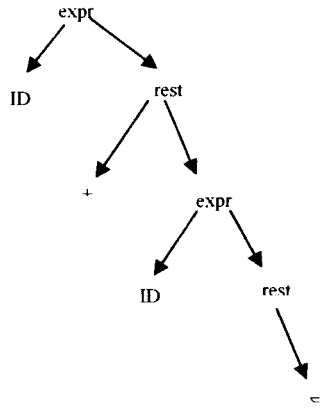
شکل ۳-۲۷ گسترش rest

با توجه به ID در ورودی ، در مرحله بعدی برای گسترش expr فقط یک انتخاب به صورت expr→ID rest وجود دارد. در نتیجه درخت تجزیه ذیل به دست می‌آید.



شکل ۳-۲۸ گسترش expr

در این مرحله برای rest سه انتخاب وجود دارد، اما با توجه به اینکه ورودی تمام شده است، rest را باید به گونه ای گسترش داد که∈ را تولید کند. برای تولید ∈ فقط یک انتخاب به صورت rest→∈ وجود دارد، در نتیجه درخت تجزیه به صورت ذیل گسترش می‌یابد.



شکل ۳-۲۹ گسترش rest

گرامر ذیل را در نظر بگیرید

S→aSb|ab

این گرامر LL(1) نیست زیرا اگر نماد جاری a باشد نمی‌توان تعیین کرد که آیا ab باید انتخاب شود یا aSb زیرا هر دو با a شروع می‌شوند.

در برخی موارد می‌توان با استفاده از k نماد رشته ورودی قاعده تولید بعدی را تعیین کرد که در این صورت گرامر را LL(k) می‌نامیم.

مثال آیا گرامر ذیل LL(2) است.

S→aSb|ab

این گرامر LL(2) است زیرا اگر دو نماد رشته ورودی را در نظر بگیریم می‌توان قاعده تولید بعدی را مشخص کرد. اگر دو نماد بعدی ab باشد S→ab انتخاب می‌شود و اگر aa باشد S→aSb انتخاب می‌گردد، در غیر این صورت رشته رد می‌شود.

گرامرهای LL(1) مبهم نیستند، در نتیجه اگر ثابت کنیم گرامری LL(1) است، ثابت می‌شود گرامر مبهم نیست. برای تشخیص LL(1) بودن گرامر می‌توان از قواعد زیر استفاده کرد.

-گرامرهای دارای بازگشتی چپ LL(1) نیستند(چرا؟).

- گرامرهای مبهم LL(1) نیستند(چرا؟).

- اگر جدول تجزیه غیربازگشتی پیشگو دارای خانه‌ای با بیش از یک وارده باشد، گرامر LL(1) نیست و به عکس اگر جدول تجزیه پیشگوی غیربازگشتی دارای خانه‌ای با بیش از یک وارده نباشد، گرامر LL(1) است. در نتیجه تولید جدول تجزیه یکی از مهمترین روشهای تست LL(1) بودن گرامر است.
- اگر در گرامر قاعده تولیدی به صورت $A\rightarrow\alpha|\beta$ وجود داشته باشد به طوریکه α و β با یک پایانه یکسان شروع شوند گرامر LL(1) نیست. به عبارت دیگر اگر رابطه ذیل برقرار باشد.

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

در این وضعیت برخورد first/first رخ می‌دهد، در نتیجه گرامر LL(1) نیست.

مثال۳-۳۷ به گرامر ذیل دقت کنید.

$A\rightarrow aB \mid aad$
$B\rightarrow bB \mid c$

با توجه به:

$first(aB)=\{a\}$
$first(aad)=\{a\}$

در نتیجه:

$first(aB) \cap first(aad)=\{a\}\neq\varnothing$

برخورد firts/first رخ داده است و بنابراین گرامر LL(1) نیست.

- اگر در گرامر، قاعده تولیدی به صورت $A\rightarrow\alpha|\beta$ وجود داشته باشد به طوریکه α و β هر دو رشته تهی را تولید کنند، گرامر LL(1) نیست.

مثال۳-۳۸ گرامر ذیل را در نظر بگیرید.

$A\rightarrow CB \mid \epsilon$
$B\rightarrow bB \mid \epsilon$
$C\rightarrow cC \mid \epsilon$

در $A\rightarrow CB \mid \epsilon$:

$\alpha=CB$
$\beta=\epsilon$

در قاعده تولید $A\rightarrow CB|\epsilon$ دو انتخاب CB و ϵ هر دو ϵ را تولید می‌کنند. در نتیجه گرامر LL(1) نیست.

- اگر در گرامر، قاعده تولیدی به صورت $A\rightarrow\alpha|\beta$ وجود داشته باشد به طوریکه β بتواند رشته تهی را تولید کند و نمادی در مجموعه‌های follow(A) و $first(\alpha)$ مشترک باشد، برخورد first/follow رخ می‌دهد در نتیجه گرامر LL(1) نیست. به عبارتی اگر رابطه ذیل برقرار باشد گرامر LL(1) نیست.

$first(\beta) \cap follow(A) \neq \varnothing$

مثال۳-۳۹ گرامر ذیل را در نظر می‌گیریم.

S→Aab
A→a | ∈

در قاعده تولید A→a|∈ ، α=a و β=∈ است. با توجه به S→Aab ، پایانه a بعد از A قرار دارد در نتیجه follow(A)={a} است. با توجه به A→a|∈، first(a)={a} است، در نتیجه:

$first(a) \cap follow(A)=\{a\} \neq \varnothing$

برخورد first/follow رخ می‌دهد و در نتیجه گرامر LL(1) نیست.

گرامرهایی که LL(1) نیستند را با اعمال تغییراتی در گرامر می‌توان به گرامر LL(1) تبدیل کرد. از جمله روشهای تبدیل گرامر غیر LL(1) به گرامر LL(1) عبارتند از:

- حذف بازگشتی چپ: همانطور که ذکر شد گرامرهای دارای بازگشتی چپ ،LL(1) نیستند. می‌توان با استفاده از روشهایی که در ابتدای فصل ذکر شد بازگشتی چپ را حذف نمود.
- فاکتورگیری چپ: برای رفع برخورد first/first می‌توان از فاکتورگیری چپ استفاده نمود.

مثال۳-۴۰ گرامر ذیل LL(1) نیست.

A→aB | aad
B→bB | c

با استفاده از فاکتورگیری چپ گرامر فوق به گرامر ذیل تبدیل می‌گردد.

A→aR
R→B | ad
B→bB | c

گرامر جدید دارای برخورد first/first نیست.

گرامرهایی وجود دارند که با هیچ تغییری نمی‌توان آنها را به LL(1) تبدیل نمود. اگر برای تعیین قاعده تولید بعدی نیاز به بررسی K نماد بعدی از ورودی باشد گرامر از نوع LL(k) است.

مثال۳-۴۱ آیا گرامر ذیل LL(1) است.

A→ aCbAB | d
B→ eA|∈
C→ c

با چند روش می‌توان LL(1) بودن این گرامر را بررسی کرد. در ذیل به دو روش می‌پردازیم.

۱- گرامر مبهم است در نتیجه LL(1) نیست.

۲- جدول تجزیه این گرامر را رسم می‌کنیم.

ابتدا first سمت راست تمام غیر پایانه‌ها را محاسبه می‌کنیم و بر اساس آن جدول تجزیه را تکمیل می‌کنیم.

جدول ۳-۶ محاسبه first

قاعده تولید	first سمت راست قاعده تولید	وارده جدول تجزیه
A→ aCbAB	first(aCbAB)={a}	M[A,a]=A→ aCbAB
A→ d	first(d)={d}	M[A,d]= A→ d
B→ eA	first(eA)={e}	B→ eA
C→ c	first(c)={c}	C→ c

با توجه به اینکه ∈ در first(B) قرار دارد و با توجه به قانون ۲ تکمیل جدول تجزیه[۱] M[B,e]=B→∈ می‌شود زیرا e در follow(B) قرار دارد. نتیجه در جدول ذیل نشان داده شده است.

جدول ۳-۷ جدول تجزیه پیشگوی غیر بازگشتی

غیر پایانه	نماد ورودی					
	d	**c**	**e**	**a**	**b**	**$**
A	A→d			A→ aCbAB		
B			B→∈ B→eA			B→∈
C		C→c				

با توجه به جدول، در M[B,e] دو قاعده تولید وجود دارد، در نتیجه گرامر LL(1) نیست.

مثال۳-۴۲ آیا گرامر ذیل LL(1) است؟

S→ Aa
S→ Bb
A→ ∈
B→ ∈
A→ cAb
B→ dAa

جدول تجزیه پیشگوی غیر بازگشتی (جدول تجزیه LL(1)) گرامر را تشکیل می‌دهیم.

جدول ۳-۸ جدول تجزیه پیشگوی غیر بازگشتی

غیر پایانه	نماد ورودی				
	a	**b**	**c**	**d**	**$**
S	S→ Aa	S→ Bb	S→ Aa	S→ Bb	
A	A→ ∈	A→ ∈	A→ cAb		
B		B→ ∈		B→ dAa	

۱. قانون ۲: اگر ∈ در first(α) باشد برای هر پایانه b در follow(A)، M[A,b]=A→∈ می‌گردد.

با توجه به جدول چون هیچ خانه ای از جدول دو قاعده تولید ندارد در نتیجه ثابت می‌شود گرامر LL(1) است.

مثال ۳-۴۳ اگر برای گرامر ذیل جدول تجزیه LL(1) را تشکیل دهیم در سطر A و ستون b چه خواهیم داشت[1]:

$S \rightarrow aAb \mid bB$
$A \rightarrow aA \mid \epsilon$

ابتدا first سمت راست قواعد تولید را محاسبه می‌کنیم.

first(aAb)={a}
first(bB)={b}
first(aA)={a}
first(∈)={∈}

مطابق قانون ۲، با توجه به $A \rightarrow \epsilon$ و وجود ∈ در first(∈) ، برای هر پایانه موجود در follow(A)={b} قاعده تولید $A \rightarrow \epsilon$ به M[A,b] اضافه می‌گردد، در نتیجه:

$M[A,b] = A \rightarrow \epsilon$

مثال ۳-۴۴ آیا گرامر ذیل LL(1) است؟

$S \rightarrow aACb$
$A \rightarrow b \mid \epsilon$
$C \rightarrow cC \mid \epsilon$

جدول تجزیه پیشگوی غیر بازگشتی (جدول تجزیه LL(1)) را تشکیل می‌دهیم.

جدول۳-۹ جدول تجزیه پیشگوی غیربازگشتی

غیر پایانه	نماد ورودی			
	a	b	c	$
S	$S \rightarrow aACb$			
A		$A \rightarrow b$ $A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B		$C \rightarrow \epsilon$	$C \rightarrow cC$	

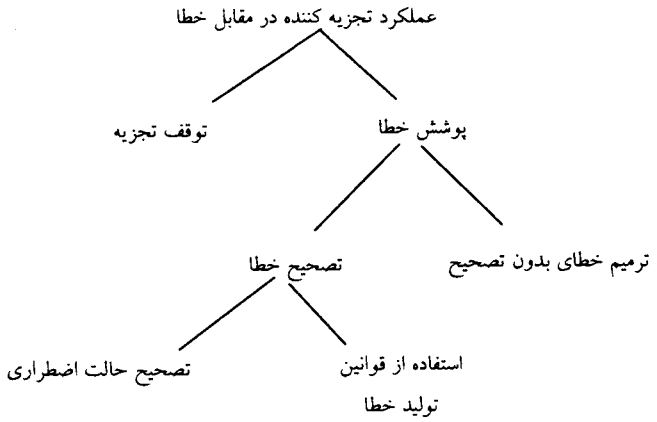
با توجه به جدول، چون M[A,b] دارای دو قاعده تولید است بنابراین گرامر LL(1) نیست.

۳-۱۵-۲ مدیریت خطا در تجزیه کننده

برای برخورد تحلیلگر نحوی با خطا روشها مختلفی وجود دارد. عملکرد تحلیلگر نحوی در برخورد با خطا را می‌توان به صورت ذیل خلاصه کرد.

---

۲. منظور از جدول تجزیه LL(1)، جدول تجزیه پیشگوی غیر بازگشتی است.



شکل ۳-۳۰ روش‌های برخورد با خطا

تحلیلگر نحوی هنگام کشف یک خطای نحوی دو راه در اختیار دارد.

۱- توقف تجزیه: در این روش تجزیه کننده، تجزیه را متوقف می‌کند و پیغام خطای مناسب را به برنامه نویس می‌دهد. بعد از رفع خطا توسط برنامه نویس، برنامه دوباره باید کامپایل شود تا خطاهای دیگر در صورت وجود کشف شود. در این روش برای هر خطا تجزیه متوقف می‌شود. این توقفها باعث طولانی شدن خطایابی و در نهایت طولانی شدن تولید برنامه می‌شود.

۲- پوشش خطا : دراین روش تجزیه کننده از خطا عبور کرده و خطاهای دیگری را نیز کشف و پیغام‌های مناسب را به برنامه نویس می‌دهد. در نتیجه برنامه نویس می‌تواند اکثر خطاها را یکجا رفع کند. عیب این روش آن است که ممکن است تجزیه کننده در کشف خطاهای بعدی دچار مشکل گردد به عبارتی خطاهایی را نادیده گرفته و یا مواردی که صحیح است را به عنوان خطا اعلام می کند. برای پوشش خطا روشهای مختلفی وجود دارد که مهمترین آنها عبارتند از:

الف- ترمیم خطای بدون تصحیح: در این روش ورودی تغییر نمی‌کند بلکه تمام اطلاعات تجزیه کننده از بین می‌رود و سپس تجزیه کننده بقیه برنامه را تجزیه می‌کند. اگر ادامه تجزیه با موفقیت ادامه یابد خطای دیگری وجود ندارد و اگر تجزیه با شکست مواجه شود، خطای دیگری وجود دارد.

ب- روش تصحیح خطا: در این روش تجزیه کننده جریان ورودی و یا تجزیه کننده را اصلاح می‌کند تا تجزیه ادامه یابد. روش تصحیح خطا، از روش ترمیم خطای بدون تصحیح معمول تر است. مهمترین روشهای تصحیح خطا عبارتند از:

۱- تصحیح حالت اضطراری[1]: در این روش تجزیه کننده هنگام کشف یک خطا، نمادهای ورودی را تا رسیدن به یک علامت هماهنگ کننده که به وسیله طراح کامپایلر معین شده است نادیده می‌گیرد. به عنوان مثال علامت سمی کالن در زبان C یک علامت هماهنگ کننده است.

۲- استفاده از قوانین تولید خطا[2]: اگر خطاهایی که در برنامه رخ می‌دهد را بتوان پیش بینی کرد می‌توان قواعد تولیدی به گرامر اضافه کرد تا این خطا را نیز شامل گردد. در این صورت تجزیه کننده با کاهش چنین گرامری خطا را کشف می‌کند و می‌تواند به تجزیه نیز ادامه دهد.

مثال ۳-٤٥ گرامر ذیل نشان می‌دهد که بعد از هر دستور سمی کالن قرار دارد.

stmt→stmt ; stmt_list | stmt ;

با توجه به آمار، یکی از خطاهایی که معمولا در برنامه رخ می‌دهد عدم درج سمی کالن است در نتیجه قاعده تولیدی به صورت ذیل نیز به گرامر اضافه می‌گردد.

stmt_error→stmt stmt_list | stmt

در نتیجه اگر برنامه نویس سمی کالن را درج نکند دستورات به جای stmt با stmt_error کاهش می‌یابد بنابراین تجزیه کننده متوقف نمی‌گردد بلکه پیغام مناسب را صادر و تجزیه ادامه می‌یابد.

۳-۱۵-۳ پوشش خطا در تجزیه کننده پیشگوی غیر بازگشتی

یکی از روشهای پوشش خطا، تصحیح خطا در حالت اضطراری است در این روش نمادهای ورودی تا زمانی که یک نماد هماهنگ کننده ظاهر گردد حذف می‌شوند. کارایی این روش به نحوه تعیین نماد هماهنگ کننده بستگی دارد. برخی از روشهای انتخاب نمادهای هماهنگ کننده عبارتند از:

۱- اگر پایانه‌ای مانند a بالای پشته باشد که با نماد جاری یکسان نباشد، خطا رخ می‌دهد، به منظور پوشش خطا، نماد a را به ورودی اضافه می‌کنیم، بنابراین نماد بالای پشته با نماد جاری یکسان می‌شود بنابراین انتقال تطبیق انجام می‌گردد و نماد بالای پشته و نماد ورودی اضافه شده حذف می‌گردد و امید می‌رود تجزیه با موفقیت ادامه یابد، به عبارت ساده نماد

---

1. Panic Mode
2. Error Production

بالای پشته حذف می‌گردد. در این روش پیغامی مبنی بر درج a به برنامه‌نویس داده خواهد شد، تا برنامه‌نویس از خطا مطلع شود.

مثال۳-٤٦ گرامر ذیل را در نظر می‌گیریم.

S→bcA
A→aA|c

جدول تجزیه این گرامر در جدول ذیل نشان داده شده است.

جدول ۳-۱۰ جدول تجزیه پیشگوی غیر بازگشتی

	a	b	c	$
S		S→bcA		
A	A→aA		A→c	

رشته bdcc را در مراحل ذیل تجزیه می‌کنیم.

جدول ۳-۱۱ مراحل تجزیه رشته bdcc

پشته	ورودی	توضیحات
$S	bdcc$	
$Acb	bdcc$	
$Ac	dcc$	نماد بالا پشته c و نماد ورودی d است، در نتیجه خطا رخ می‌دهد. بنابراین به منظور پوشش خطاc را به ورودی اضافه می‌کنیم.
$Ac	cdcc$	نماد بالای پشته و نماد ورودی یکسان است و در نتیجه نماد بالای پشته و نماد ورودی حذف می‌شوند و تجزیه ادامه می‌یابد.

جدول ۳-۱۱ مفهوم پوشش خطا را نشان می‌دهد. زیرا علی رغم وجود خطا در ورودی تجزیه ادامه می‌یابد، و این مفهوم پوشش خطا است.

۲- نمادهای first(A) را به عنوان مجموعه هماهنگ کننده A در نظر می‌گیریم . بنابراین اگر یکی از نمادهای first(A) در ورودی ظاهر شود تجزیه بر اساس A ادامه می‌یابد.

مثال ۳-٤٧ گرامر ذیل را در نظر می‌گیریم.

S→bA
A→aA|c

جدول تجزیه این گرامر در جدول ذیل نشان داده شده است.

جدول ۳-۱۲ جدول تجزیه پیشگوی غیر بازگشتی

	a	b	c	$
S		S→bA		
A	A→aA		A→c	

مراحل تجزیه رشته bdec در جدول ذیل نشان داده شده است.

جدول ۳-۱۳ مراحل تجزیه رشته bdec

پشته	ورودی	توضیحات
$S	bdec$	
$Ab	bdec$	
$A	dec$	M[A,d] خالی است در نتیجه خطا رخ داده است. بنابراین از نمادهای ورودی تا رسیدن به نمادی از first(A) صرفنظر می‌شود. در نتیجه از d صرفنظر می‌شود.
$A	ec$	M[A,e] خالی است در نتیجه خطا رخ داده است. بنابراین از نمادهای ورودی تا رسیدن به نمادی از first(A) صرفنظر می‌شود. بنابراین از e صرفنظر می‌شود.
$A	c$	
$c	c$	
$	$	

۳- نمادهای follow(A) را به عنوان نمادهای هماهنگ کننده A در نظر می‌گیریم. follow(A) نمادهایی هستند که بعد از A قرار می‌گیرند، بنابراین از نمادهای ورودی تا رویت یکی از نمادهای follow(A) صرفنظر می‌کنیم و پس از رسیدن به یکی از نمادهای follow(A) ،غیر پایانه A را از بالای پشته حذف می‌کنیم.

مثال ۳-۴۸ گرامر ذیل را در نظر می‌گیریم.

S→bAe
A→aA|c

جدول تجزیه این گرامر در جدول ذیل نشان داده شده است.

جدول۳-۱٤ جدول تجزیه پیشگوی غیر بازگشتی

	a	b	c	e	$
S		S→bAe			
A	A→aA		A→c		

مراحل تجزیه رشته bde در جدول ذیل نشان داده شده است.

جدول ۳-۱۵ مراحل تجزیه رشته bde

پشته	ورودی	توضیحات
$S	bde$	
$eAb	bde$	
$eA	de$	M[A,d] خالی است در نتیجه خطا رخ داده است. بنابراین از نمادهای ورودی تا رسیدن به نمادی از follow(A) صرفنظر می‌شود. بنابراین از d صرفنظر می‌شود.
$eA	e$	M[A,e] خالی است در نتیجه خطا رخ داده است. بنابراین از نمادهای ورودی تا رسیدن به نمادی از follow(A) صرفنظر می‌شود. چون e در follow(A) است در نتیجه A از پشته حذف می‌شود.
$e	e$	
$	$	

این روش معایبی نیز دارد به عنوان مثال به قطعه برنامه ذیل دقت کنید:

```
a:=b+c
if(a=b)
   d=d+;
```

دستور اول علامت ; از قلم افتاده است بنابراین یک خطای نحوی رخ داده است. در نتیجه تجزیه کننده تا رسیدن به علامت ; که علامت هماهنگ کننده است ورودی را حذف می‌کند ، بنابراین کل عبارت if (a=b) d=d+1 حذف می‌گردد.

٤- برای رفع مشکل روش ۳ کلمات کلیدی شروع کننده دستورات را به مجموعه هماهنگ کننده اضافه می‌کنیم. به عنوان مثال کلمات repeat،while،if را به مجموعه هماهنگ کننده غیر پایانه تولید کننده عبارات اضافه می‌کنیم.

۳-۱٦ تجزیه کننده پایین به بالا

تجزیه کننده پایین به بالا ساخت درخت تجزیه را از برگها شروع می‌کند و به سمت ریشه پیش می‌رود اگر تجزیه کننده بتواند درخت تجزیه را برای رشته ورودی بسازد، رشته پذیرفته می‌گردد در غیر این صورت رشته ورودی رد می‌شود. تجزیه کننده پایین به بالا عکس عمل اشتقاق را انجام می دهد. اشتقاق از نماد شروع، رشته را تولید می‌کند، ولی تجزیه کننده پایین به بالا رشته ورودی را به نماد شروع کاهش می‌دهد. اشتقاق در هر مرحله با استفاده از قاعده تولید A→α سمت چپ A را به سمت راست α گسترش می‌دهد در حالیکه تجزیه کننده پایین به بالا در هر مرحله دنباله α را به سمت چپ A کاهش می‌دهد.

مثال۳-٤۹ گرامر ذیل را در نظر بگیرید.

expr→ expr + term | expr - term | term
term→ 1|2|3|4

رشته 4+1-2 توسط تجزیه کننده پایین به بالا به صورت ذیل تجزیه می‌گردد.

```
4+1-2
term +1-2
expr + 1 -2
expr + term -2
expr - 2
expr - term
expr
```

رشته 4+1-2 با موفقیت به نماد شروع کاهش می‌یابد، در نتیجه این رشته پذیرفته می‌شود. اما رشته 1+2-+3 با توجه به مراحل ذیل قابل کاهش به نماد شروع نیست.

```
1+2-+3
term+ 2-+3
expr + 2 -+ 3
expr + term  -+3
expr -+ 3
expr -+ term
expr -+  expr
```

تجزیه کننده قادر به کاهش رشته 1+2-+3 به نماد شروع نیست، در نتیجه رشته 4+2-+3 رد می‌شود. رشته 4+1-2 را به روشهای مختلفی می‌توان به نماد شروع کاهش داد. که برخی از این روشها در جدول ۳-۱۶ ارائه شده است.

همانطور که ملاحظه می‌گردد به روشهای مختلف می‌توان رشته 4+1-2 را به نماد شروع کاهش داد. اگر این روند به گونه‌ای درست انجام نگردد ممکن است مانند روش سوم رشته ورودی به نماد شروع منجر نگردد و تجزیه با شکست مواجه شود.

جدول ۳-۱٦ انواع روشهای کاهش رشته 4+1-2 به نماد شروع

روش اول	روش دوم
4+1-2 4+ term -2 4+ term -term term + term -term expr + term -term expr - term expr	4+1-2 term +1-2 expr + 1 -2 expr + term -2 expr - 2 expr - term expr
روش سوم	**روش چهارم**
4+1-2 4+ 1 - term 4+ term -term term + expr -term term + expr (عدم موفقیت در تجزیه)	4+1-2 4+ 1 - term 4+ term -term term + term -term expr + term -term expr - term expr

اگر ترتیب کاهش رشته ورودی به نماد شروع، معکوس سمت راست ترین اشتقاق باشد، دنباله‌ای که در هر مرحله کاهش می‌یابد، دستگیره[1] می‌نامیم. به عنوان مثال روش دوم کاهش رشته 4+1-2 که در جدول ۳-۱۶ ارائه شد، را در نظر می‌گیریم. مراحل کاهش و ترتیب عکس آن که در ذیل ارائه شده است را در نظر می‌گیریم.

جدول ۳-۱۷ مقایسه تجزیه پایین به بالا با سمت‌راست ترین اشتقاق

تجزیه پایین به بالا	سمت راست ترین اشتقاق
4+1-2	expr
term +1-2	expr - term
expr + 1 -2	expr - 2
expr + term -2	expr + term - 2
expr - 2	expr + 1 - 2
expr - term	term + 1 - 2
expr	4 + 1 - 2

همانطور که جدول فوق نشان می‌دهد معکوس ترتیب کاهش‌ها نشان دهنده سمت راست ترین اشتقاق است. در نتیجه هر دنباله‌ای که کاهش می‌یابد یک دستگیره است. دستگیره‌ها در کاهش رشته 4+1-2 در جدول ذیل نشان داده شده است.

جدول ۳-۱۸ دستگیره‌ها در تجزیه پایین به بالا رشته 4+1-2

دستگیره	کاهش‌ها	قاعده تولید مورد استفاده
4	4+1-2	term→4
term	term +1-2	expr→term
1	expr + 1 -2	term→1
expr + term	expr + term -2	expr→expr+term
2	expr - 2	term→2
expr - term	expr - term	expr→expr+term
	expr	

مثال۳-۵۰ دستگیره‌ها را در کاهش رشته bccdef با توجه به گرامر ذیل مشخص کنید.

S→bBCf
B→Bcd|c
C→e

ابتدا رشته bccdef را بوسیله سمت راست ترین اشتقاق تولید می‌کنیم.

S
bBCf
bBef
bBcdef
bccdef

1. Handle

عکس ترتیب به سمت راست‌ترین اشتقاق را در نظر می‌گیریم. دنباله‌ای که در هر مرحله کاهش می‌یابد، یک دستگیره است. دستگیرها در جدول ۳-۱۹ نشان داده شده است.

جدول ۳-۱۹ دستگیره‌های کاهش رشته bccdef

کاهش	دستگیره
bccdef	c
bBcdef	Bcd
bBef	e
bBCf	bBCf
S	

با توجه رابطه دستگیره و سمت راست ترین اشتقاق می‌توان دستگیره را به صورت ذیل نیز تعریف کرد.

دستگیره، دنباله‌ای است که منطبق بر سمت راست یک قاعده تولید بوده و کاهش آن به سمت چپ قاعده تولید یک مرحله از مراحل معکوس سمت راست‌ترین اشتقاق است. با توجه به تعاریف ذکر شده دنباله‌هایی که در روشهای دیگر کاهش غیر از عکس سمت راست‌ترین اشتقاق کاهش می‌یابد مانند دنباله‌هایی که در روشهای دوم، سوم، چهارم کاهش رشته 2-1+4 به دست می‌آید دستگیره نیستند. زیرا اگر آنها را به ترتیب عکس بخوانیم منجر به سمت راست‌ترین اشتقاق نخواهد شد. اگر گرامر مبهم نباشد در هر مرحله فقط یک دستگیره یافت می شود و اگر گرامر مبهم باشد در یک مرحله چند دستگیره ممکن است یافت شود.

با توجه به مفهوم دستگیره، تجزیه کننده‌های پایین به بالا دو مرحله اصلی ذیل را تکرار می‌کنند.

۱- یافتن دستگیره

۲- کاهش دستگیره

همه تجزیه کننده‌های پایین به بالا بعد از یافتن دستگیره، آن را کاهش می‌دهند. تفاوت انواع تجزیه کننده‌های پایین به بالا در نحوه کشف دستگیره‌ها است. مهمترین تجزیه کننده‌های پایین به بالا عبارتند از:

۱- تجزیه کننده عملگر- اولویت

۲- تجزیه کننده‌های LR

تفاوت این تجزیه کننده در قدرت آنها در کشف دستگیره‌ها است. به عنوان مثال یک نوع تجزیه کننده برای یک گرامر به خصوص قادر به تعیین دستگیره نیست در حالیکه تجزیه کننده دیگر ممکن است بتواند دستگیره را تعیین کند. در ادامه به بررسی هر یک از انواع تجزیه کننده‌های پایین به بالا می‌پردازیم.

۳-۱۷ تجزیه کننده عملگر-اولویت

تجزیه کننده عملگر-اولویت، را می‌توان به سادگی به صورت دستی ایجاد کرد. این تجزیه کننده ضعیف است و بیشتر برای عبارات محاسباتی کاربرد دارد. تجزیه کننده عملگر-اولویت از روابط عملگرها برای تعیین دستگیره‌ها استفاده می کند. تجزیه کننده عملگر-اولویت را برای همه انواع گرامرها نمی‌توان ایجاد کرد. این تجزیه کننده فقط برای گرامرهای موسوم به گرامرهای عملگر قابل تولید است. گرامرهای عملگر، گرامرهایی هستند که دو ویژگی ذیل را داشته باشند.

۱- سمت راست هیچ قاعده تولیدی ∈ نباشد.

۲- در سمت راست هیچ قاعده تولیدی بیش از یک غیر پایانه مجاور هم وجود نداشته باشد.

مثال ۳-۵۱ گرامر ذیل یک گرامر عملگر نیست زیرا ، یکی از قواعد تولید در سمت راست ∈ دارد.

$S \rightarrow ACD$
$A \rightarrow Sc \mid D$
$B \rightarrow Dd \mid \in$

مثال ۳-۵۲ گرامر ذیل، گرامر عملگر نیست، زیرا دو غیر پایانه مجاور یکدیگر در سمت راست S قرار دارند.

$S \rightarrow AB$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid b$

گرامری که از نوع عملگر نیست را می‌توان به گرامر عملگر تبدیل کرد. برای تبدیل یک گرامر غیر عملگر به گرامر عملگر می‌توان از جایگذاری استفاده کرد.

مثال ۳-۵۳ گرامر مثال ۳-۵۲ را می‌توان به گرامر عملگر ذیل تبدیل کرد.

$S \rightarrow aAbB \mid aAb \mid abB \mid ab$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid b$

در تجزیه عملگر-اولویت بین پایانه‌ها سه رابطه $<$ ،$>$ و $=$ تعریف می‌شود مفهوم این علائم عبارتند از:

$a < b$: اولویت a از b کمتر است

a > b : اولویت a از b بیشتر است

a = b : اولویت a برابر b است

اگر چه عملگرهای >و< و = مشابه عملگرهای ریاضی <,> و = هستند ولی تفاوت اساسی با آنها دارند. یکی از مهمترین تفاوتهای آن این است که الزاما نباید یکی از رابطه‌های >و< و = برقرار باشد. به عبارتی در یک زبان، ممکن است هم > و< هر دو موجود باشد و یا هیچ یک برقرار نباشد. تعیین رابطه اولویت بین پایانه‌ها به زبان مورد نظر بستگی دارد. برخی مواردیکه برای تعیین اولویت پایانه‌ها مورد استفاده است به قرار ذیل است.

- روابط شرکت پذیری
- تقدم عملگرها

از روابط اولویت می‌توان برای تعیین دستگیره‌ها استفاده کرد . به این منظور به ابتدا و انتهای رشته ورودی w، علامت $ را اضافه می‌کنیم در نتیجه رشته w به w تبدیل می‌گردد.

اولویت $ از همه پایانه‌ها کمتر است. یک دستگیره اولین دنباله‌ای است که در پویش از چپ به راست ورودی بین< در سمت راست و> در سمت چپ محصور است. به طور متوالی دستگیره‌ها را تشخیص داده و کاهش می‌دهیم، تا به نماد شروع برسیم و به این ترتیب تجزیه انجام می‌گردد.

الگوریتم تجزیه عملگر- اولویت را می‌توان به صورت ذیل خلاصه کرد.

۱- درج $ در ابتدا و انتهای رشته ورودی

۲- درج روابط عملگر بین پایانه‌های رشته ورودی

۳- پویش چپ به راست رشته ورودی و کشف اولین دنباله محصور بین < در سمت راست و> در سمت چپ، این دنباله با غیر پایانه‌های همجوار دستگیره است. برای یافتن این دنباله رشته ورودی را از چپ به راست پویش می‌کنیم، پس از یافتن اولین < به سمت چپ پویش را ادامه می‌دهیم تا > را بیابیم. در این روند علامت = بی‌تاثیر است.

٤- کاهش دستگیره به سمت چپ قاعده تولید.

اگر نماد شروع ایجاد شود و رشته ورودی نیز تمام شود، رشته ورودی پذیرفته می‌گردد و الگوریتم پایان می‌یابد.

۵- حذف غیر پایانه‌ها و بازگشت به مرحله دو

برای درک بهتر عملکرد تجزیه کننده عملگر-اولویت گرامر ذیل را در نظر می‌گیریم.

$E \rightarrow E+E \mid E*E \mid id$

اولویت عملگرها در جدول ذیل نشان داده شده است.

جدول ۳-۲۰ جدول اولویت عملگرها

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

در جدول ۳-۲۰ سطرها، نشان دهنده پایانه سمت چپ و ستون‌ها، نشان دهنده پایانه سمت راست است. به عنوان مثال رابطه بین id و + در رشته id+id را تعیین می‌کنیم. id اول سمت چپ و + سمت راست آن است. در نتیجه برای استخراج عملگر بین id و + سطر id و ستون + را استخراج می‌کنیم. در نتیجه:

id.> +

در رشته id+ پایانه id دوم سمت راست و+ سمت چپ است. در نتیجه برای استخراج عملگر بین id و + ، سطر + و ستون id را استخراج می‌کنیم، در نتیجه:

id.> + <.id

به عنوان مثال رشته id+id*id را با استفاده از گرامر فوق تجزیه می‌کنیم.

برای تجزیه رشته id+id*id مراحل ذیل با توجه به الگوریتم انجام می‌شود.

- ابتدا و انتهای رشتهid+id*id علامت $ را درج می‌کنیم. در نتیجه رشته id+id*id به $id+id*id$ تبدیل می‌گردد.

- بین همه پایانه‌های $id+id*id$ روابط اولویت را با استفاده از جدول عملگر اولویتها درج می‌کنیم. نتیجه درج روابط به صورت ذیل خواهد شد.

$ < id > + < id > * < id > $

- اولین دنباله محصور بین < و > که در پویش چپ به راست کشف می‌شود را می‌یابیم ، اولین دنباله محصور id است. در نتیجه id اولین دستگیره است.

- id را به E کاهش می‌دهیم. رشته $id+id*id$ به $E+id*id$ تبدیل می‌شود.

- غیر پایانه‌ها را از $E+id*id$ حذف می‌کنیم در نتیجه $E+id*id$ به $+id*id$ تبدیل می‌شود.

- روابط عملگرها را در $+id*id$ درج می‌کنیم. نتیجه درج روابط به صورت ذیل است.

$ < + < id > * < id > $

- اولین دنباله محصور بین < و > را در پویش چپ به راست کشف می‌کنیم اولین دنباله، id است. در نتیجه id نیز یک دستگیره است.

- id را به E کاهش می‌دهیم. در نتیجه $E+id*id$ به $E+E*id$ تبدیل می‌گردد.
- غیر پایانه‌ها را حذف، می‌کنیم. در نتیجه رشته $E+E*id$ به $+*id$ تبدیل می‌گردد.
- بین عملگرهای $+*id$ روابط اولویت را درج می‌کنیم.

$ < + < * < id > $

- اولین دنباله محصور بین < و > را در پویش چپ به راست کشف می‌کنیم. اولین دنباله محصور id است. در نتیجه id اولین دستگیره است.
- id را به E کاهش می‌دهیم. در نتیجه دنباله $E+E*id$ به $E+E*E$ تبدیل می‌گردد.
- از $E+E*E$ غیر پایانه‌ها را حذف می‌کنیم. در نتیجه $E+E*E$ به $*+$ تبدیل می‌شود.
- برای کشف دستگیره بعدی، عملگرهای رابطه ای را در $*+$ درج می‌کنیم.

$ < + < * > $

- با توجه به عبارت حاصل، اولین عبارت محصور بین > و < دنباله * است. با توجه به رشته $E+E*E$ دوطرف * غیر پایانه E وجود داشت در نتیجه دستگیره بعدی E*E است.
- E*E به E کاهش می‌یابد. در نتیجه $E+E*E$ به $E+E$ تبدیل می‌گردد.
- برای یافتن دستگیره بعدی غیر پایانه‌ها را حذف می‌کنیم. در نتیجه دنباله $E+E$ به $+$ تبدیل می‌شود.
- روابط عملگر را در $+$ درج می‌کنیم.

$ < + > $

- اولین دنباله محصوربین< و > نماد + است. در نتیجه + همراه غیر پایانه‌های طرفین آن یعنی E+E دستگیره است.
- E+E به E کاهش می‌یابد. نماد شروع ایجاد شده است و رشته ورودی تمام شده است در نتیجه تجزیه تکمیل می‌گردد و رشته پذیرفته می‌گردد.

برای پیاده سازی این روش می‌توان از یک پشته استفاده کرد.
اگر a نماد جاری ورودی و s نماد بالای پشته باشد، از قوانین ذیل استفاده می‌کنیم.

ورودی پشته

- اگر s < a باشد، عمل انتقال a به پشته انجام می‌شود.
- اگر s = a باشد، عمل انتقال a به پشته انجام می‌شود.
- اگر s > a باشد، کاهش انجام می‌شود. در این شرایط دستگیره در پشته است. برای مشخص کردن دستگره آنقدر نمادها را از پشته حذف می‌کنیم تا اولویت نماد بالای پشته از

اولویت نماد جاری ورودی کمتر شود. آنچه از پشته حذف می‌شود همراه غیر پایانه‌های آن دستگیره است.

- اگر هیچ عملگری بین a و s تعریف نشده باشد خطای نحوی رخ داده است.
- تجزیه وقتی تمام می‌شود که نماد شروع تولید شده و رشته ورودی تمام گردد.

مثال۳-٥٤ رشته id+id*id را با استفاده از گرامر ذیل تجزیه می‌کنیم.

$E \rightarrow E+E \mid E*E \mid id$

مراحل تجزیه در جدول ذیل نشان داده شده است.

جدول ۳-۲۱ مراحل تجزیه رشته id+id*id

دستگیره	عمل	ورودی	رابطه نماد جاری ورودی و نماد بالای پشته	پشته
	انتقال به پشته	$ id + id * id $	<	$
id دستگیره است و کاهش با E→id	کاهش	$ + id * id $	>	$id
	انتقال به پشته	$ id * id $	<	$+
id دستگیره است و کاهش با E→id	کاهش	$ * id $	>	$+id
	انتقال به پشته	$ * id $	<	$+
	انتقال به پشته	$ id $	<	$+*
id دستگیره است و کاهش با E→id	کاهش	$	>	$+*id
* به همراه غیرپایانه‌های طرفین آن که در مراحل قبلی به دست آمده است. دستگیره است و کاهش با E→E*E		$	>	$+*
+ به همراه غیرپایانه‌های طرفین آن که در مراحل قبلی به دست آمده است. دستگیره است و کاهش با E→E+E		$	>	$+
رشته ورودی تمام شده و نماد شروع تولید شده است در نتیجه تجزیه تکمیل می‌گردد.		$		$

۳-۱۸ تجزیه کننده‌های LR

تجزیه کننده‌های LR رشته ورودی را از چپ به راست[1] پویش و معکوس سمت راست‌ترین اشتقاق را ایجاد می‌کنند[2]. پایان رشته ورودی باید علامت خاصی قرار داشته باشد تا تجزیه کننده LR با رسیدن به آن، پایان رشته را تشخیص دهد. این علامت نباید از پایانه‌های گرامر باشد. به عنوان مثال می‌توان از علامت EOF به عنوان علامت پایان رشته استفاده کرد، علامت EOF، وقتی رشته در فایل است در پایان فایل قرار دارد، و یا از علامت \n یا \0 می‌توان برای نشان دادن پایان رشته نیز استفاده کرد. اینکه چه کاراکتری را برای پایان فایل استفاده کنیم تاثیری در عملکرد تجزیه کننده ندارد. در ادامه فصل از $ به عنوان پایان رشته استفاده می‌کنیم. در نتیجه $ نشان دهنده پایان رشته ورودی است. در ساخت تجزیه کننده LR به هر گرامر یک قاعده تولید به صورت ذیل اضافه می‌کنیم.

غیر پایانه شروع گرامر → غیر پایانه جدید

اگر تجزیه کننده موفق به کاهش این قاعده تولید شود و نماد ورودی $ باشد، غیر پایانه شروع گرامر کاهش یافته است و رشته ورودی نیز تمام شده است، زیرا $ نشان دهنده پایان رشته ورودی است، در نتیجه با کاهش قاعده تولید جدید و رویت $ در ورودی، تجزیه با موفقیت انجام شده است و رشته پذیرفته می‌گردد.

مثال۳-۵۵ گرامر ذیل را در نظر می‌گیریم.

E→E+T
E→T
T→id

قاعده تولید S→E را به گرامر فوق اضافه می‌کنیم. گرامر به صورت زیر تبدیل می‌گردد.

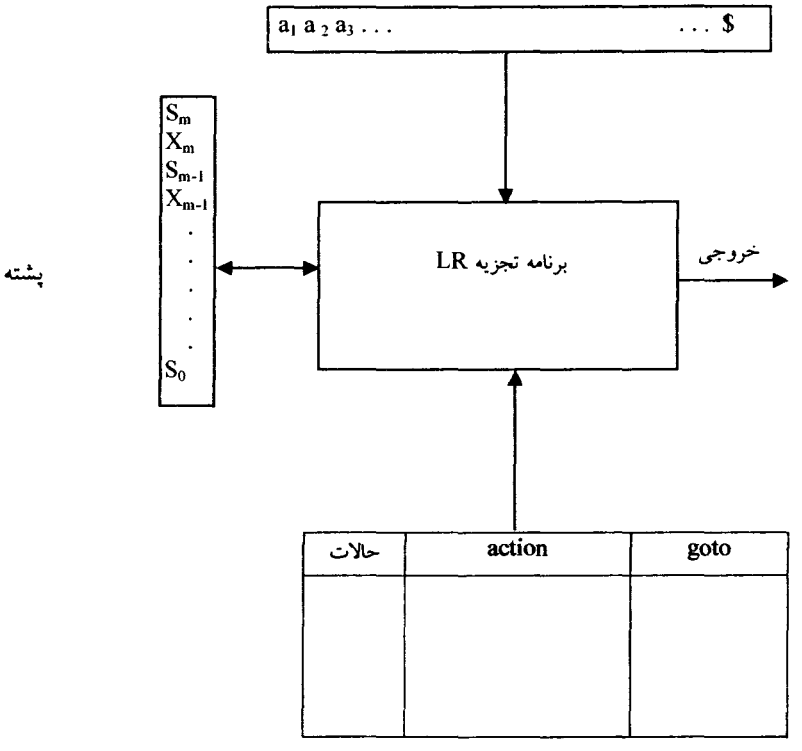
S → E
E→E+T
E→T
T→id

هرگاه تجزیه کننده موفق به کاهش S→E گردد، و نماد ورودی نیز $ باشد، تجزیه کامل شده است زیرا نماد شروع تولید ورشته ورودی تمام شده است. تجزیه کننده‌های LR ، دستگیره‌ها را کشف و سپس کاهش می‌دهند. این نوع تجزیه کننده‌ها برای کشف دستگیره‌ها از پشته استفاده می‌کنند. شکل ۳-۳۱ قسمتهای مختلف یک تجزیه کننده LR را نشان می‌دهد.

---

1. Left to right

۲. LR مخفف عبارات Left to right و Rightmost derivation است.



شکل ۳-۳۱ قسمتهای مختلف تجزیه کننده LR

قسمتهای مختلف این تجزیه کننده عبارتند از:

۱- ورودی: رشته ورودی که تجزیه کننده باید آن را تجزیه کند در این قسمت قرار دارد. به انتهای رشته ورودی $ اضافه می‌شود. تجزیه کننده LR با دیدن $ پایان رشته ورودی را تشخیص می‌دهد.

۲- پشته: محتوای پشته به صورت $S_0X_1S_1...X_mS_m$ نشان داده می‌شود. Si نشان دهنده حالات و X_i پایانه‌ها و غیر پایانه‌های گرامر است. نمادهای ورودی به پشته منتقل[1] می‌شوند تا یک دستگیره در پشته یافت شود، پس از کشف دستگیره، کاهش[2] انجام می‌شود.

1. Shift
2. Reduce

۳- جدول تجزیه LR: این بخش شامل دو قسمت action و goto است. قسمت action عملی که باید انجام شود و بخش goto حالت بعدی را مشخص می‌کند.

٤- خروجی: دنباله ای از قواعد تولید گرامر است که در تجزیه استفاده شده اند.

٥- برنامه تجزیه کننده LR: این برنامه قسمت اصلی تجزیه کننده است. برنامه تجزیه کننده بر اساس نماد جاری رشته ورودی و حالت بالای پشته و محتوای جدول تجزیه، مرحله بعدی را تعیین می‌کند. نحوه عملکرد برنامه تجزیه کننده را با یک مثال نشان می‌دهیم.

مثال۳-٥٦ گرامر ذیل را در نظر می‌گیریم.

E→E+T
E→T
T→id

قاعده تولید S→E را به گرامر اضافه می‌کنیم و برای اختصار به هر قاعده تولید یک شماره تخصیص می‌دهیم. در نتیجه گرامر به صورت ذیل است.

1- S → E
2- E→E+T
3- E→T
4- T→id

جدول تجزیه گرامر، در جدول ۳-۲۲ نشان داده شده است.

جدول۳-۲۲ جدول تجزیه LR

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	r4	r4	r4		
2	r3	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5	r2	r2	r2		

.نحوه ساخت جدول تجزیه در بخشهای بعدی تشریح خواهد شد. در جدول تجزیه s اختصار shift و r اختصار کلمه reduce است. در ابتدا علامت $ به انتهای رشته ورودی و $ و حالت اولیه به پشته اضافه می‌گردد. اگر نماد جاری رشته ورودی a و حالت بالای پشته s باشد، برنامه تجزیه LR با توجه به محتوای جدول اعمال ذیل را انجام می‌دهد.

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

۲-کاهش: اگر action[i,a]=r n باشد، یک دستگیره یافت شده است که باید کاهش یابد. اگر n شماره قاعده تولید A→β باشد، برنامه تجزیه کننده، دنباله β را از بالای پشته حذف می‌کند.

پس از حذف β اگر حالت بالای پشته m باشد، برنامه تجزیه کننده ابتدا A و سپس، عدد موجود در goto[m,A] را به بالای پشته اضافه می‌کند.

لازم به ذکر است که برای حذف β از بالای پشته تعداد 2|β| (|β|تعداد نمادها و یا طول β است) از بالای پشته حذف می‌گردد. زیرا بین هر نماد در پشته یک شماره حالت نیز وجود دارد در نتیجه برای حذف |β|نماد از بالای پشته، به تعداد |β|حالت نیز حذف می‌گردد.

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

٤- خطا: اگر action[i,a]=error باشد، تجزیه رشته ورودی با عدم موفقیت روبرو شده است.

برای درک بهتر نحوه استفاده از جدول تجزیه با استفاده از قوانین فوق رشته id+id را تجزیه می‌کنیم. مراحل تجزیه در جدول ذیل نشان داده شده است.

جدول ۳-۲۳ مراحل تجزیه رشته id+id

پشته	رشته ورودی	خروجی	توضیحات
0	id+id$		با توجه به action[0,id]=s1، تجزیه کننده id و سپس 1 را به پشته منتقل می‌کند، نتیجه این تغییر در مرحله بعد نشان داده شده است.
0id1	+id$	T→id	با توجه به action[1,+]=r4 و با توجه به اینکه قاعده تولید شماره 4 قاعده تولید T→id است، تجزیه کننده id و 1 را از بالای پشته حذف می‌کند. در نتیجه حالت بالای پشته 0 است. با توجه به goto[0,T]=2 ، تجزیه کننده ابتدا T وسپس 2 را به پشته اضافه می‌کند. در این مرحله id دستگیره است. نتیجه این تغییر در مرحله بعد نشان داده شده است.
0T2	+id$	E→T	با توجه به action[2,+]=r3 ، و با توجه به اینکه قاعده تولید شماره 3 قاعده تولید E→T است، تجزیه کننده T و 2 را از بالای پشته حذف می‌کند. در نتیجه حالت بالای پشته 0 است. با توجه به goto[0,E]=3 ، تجزیه کننده ابتدا E وسپس 3 را به پشته اضافه می‌کند. در این مرحله T دستگیره است. نتیجه این تغییر در مرحله بعد نشان داده شده است.
0E3	+id$		با توجه به action[3,+]=s4، تجزیه کننده + و سپس 4 را به پشته منتقل می‌کند، نتیجه این تغییر در مرحله بعد نشان داده شده است.

0E3+4	id$		با توجه به action[4,id]=s1، تجزیه کننده id و سپس 1 را به پشته منتقل می‌کند. نتیجه این تغییر در مرحله بعد نشان داده شده است.
0E3+4id1	$	T→id	با توجه به action[1,$]=r4 ، و با توجه به اینکه قاعده تولید شماره 4 قاعده تولید T→id است، تجزیه کننده id و 1 را از بالای پشته حذف می‌کند. در نتیجه حالت بالای پشته 4 است. با توجه به goto[4,T]=5 ، تجزیه کننده ابتدا T وسپس 5 را به پشته اضافه می‌کند. در این مرحله id دستگیره است. نتیجه این تغییر در مرحله بعد نشان داده شده است.
0E3+4T5	$	E→E+T	با توجه به action[5,$]=r2 ، و با توجه به اینکه قاعده تولید شماره 2 قاعده تولید E→E+T است، تجزیه کنندهE+T و اعداد بین این نمادها یعنی 5,4,3 را حذف می‌کند. در نتیجه حالت بالای پشته 0 است. با توجه به goto[0,E]=3 ، تجزیه کننده ابتدا E وسپس 3 را به پشته اضافه می‌کند. در این مرحلهE+T دستگیره است. نتیجه این تغییر در مرحله بعد نشان داده شده است.
0E3	$		با توجه به action[3,$]=accept تجزیه کننده رشته را می‌پذیرد.

قواعد تولید ستون خروجی جدول ۳-۲۳ نشان می‌دهد تجزیه کننده به ترتیب از چه قواعد تولیدی برای تجزیه رشته ورودی استفاده کرده است. دستگیره‌ها سمت راست قواعد تولید ستون خروجی جدول ۳-۲۳ هستند. اگر قواعد تولید ارائه شده در ستون خروجی جدول ۳-۲۳ را بررسی کنیم، دستگیره‌ها مشخص شوند. دستگیره‌های مثال ۳-۵۶ در جدول ذیل نشان داده شده است.

جدول ۳-۲۴ دستگیره‌ها

رشته	قواعد تولید خروجی تجزیه کننده LR	دستگیره
id+id	T→id	id
T+id	E→T	T
E+id	T→id	id
E+T	E→E+T	E+T
E		

همانطور که جدول بالا نشان می‌دهد تجزیه کننده دستگیره‌ها را کشف و کاهش می‌دهد تا در نهایت نماد شروع ایجاد گردد و رشته ورودی تمام شود. مهمترین قسمت تجزیه کننده LR ، جدول تجزیه است.

برای ساخت جدول تجزیه روشهای گوناگونی وجود دارد. تفاوت این روشها در انواع گرامرهایی است که می‌توانند جدول تجزیه آن را تولید کنند. به عنوان مثال برای یک گرامر مشخص ممکن است برخی از این روشها جدول تجزیه آن را بسازند در حالیکه روش دیگر ممکن است قادر به تولید جدول تجزیه آن نباشد. مهمترین روشهای موجود عبارتند از:

۱- LR(0) : ساده ترین و ضعیفترین روش است.

۲- SLR(1) [1] (LR ساده): این روش از LR(0) قویتر است.

۳- LR(1) یا LR متعارف[2]: قویترین روش ساخت جدول تجزیه LR است.

٤- LALR(1): از SLR(1) قویتر و از LR(1) ضعیفتر است.

در ادامه به بررسی هر یک از روشهای ساخت جدول تجزیه می‌پردازیم.

۳-۱۸-۱- روش LR(0)

به منظور تشریح نحوه ساخت جدول تجزیه به روش LR(0)، عنصر LR که در ساخت جدول تجزیه به کار می‌رود، را معرفی می‌کنیم. عنصر LR یک قاعده تولید است که نقطه ای در سمت راست آن قرار دارد(مانند: S→α.β). مکان نقطه در عنصر LR میزان پیشرفت، در کاهش غیر پایانه سمت چپ(مانند S) را نشان می‌دهد.

عنصر S→.αβ نشان می‌دهد که برای کاهش S نیاز به کاهش α است زیرا علامت نقطه قبل از α است در نتیجه تاکنون پیشرفتی در کاهش S نداشته‌ایم. با ادامه روند تجزیه، اگر موفق به کاهش α شویم، این پیشرفت را با عنصر، S→α.β نشان می‌دهیم. عنصر S→α.β نشان می‌دهد که α کاهش یافته است و هم اکنون سعی در کاهشβ داریم. اگر β نیز کاهش یابد، نقطه از β عبور می‌کند این پیشرفت بوسیله عنصر .S→αβ نشان داده می‌شود. پس از کاهش β ،می‌توان αβ را به S را کاهش داد. عنصر .S→αβ نشان می‌دهد، یک دستگیره که همان αβ است، یافت شده است.

یک قاعده تولید S→XYZ می‌تواند چهار عنصر LR به صورت ذیل داشته باشد.

S→.XYZ
S→X.YZ

---

1. Simple LR
2. Canonical LR(CLR)

S→XY.Z
S→XYZ.

اگر S→∈ باشد عنصر LR آن .→S است. هر عنصر LR، احتمال یا فرضیه ای در مورد یافتن دستگیره ارائه می‌دهد. به عنوان مثال عنصر S→X.YZ نشان می‌دهد که X کاهش یافته است و اکنون باید Y کاهش یابد در نتیجه عنصر S→X.YZ این فرضیه را مطرح می‌کند که دستگیره بعدی مربوط به کاهش Y است. عنصر .S→XYZ نشان می‌دهد که دستگیره XYZ یافت شده است. با توجه به مفهوم عنصرLR، می‌توان عناصر LR را به دو دسته تقسیم کرد که عبارتند از:

عنصر کاهشی: عنصری که نقطه در آخر سمت راست قاعده تولید قرار دارد. این نوع عنصر نشان دهنده یافتن یک دستگیره است. به عنوان مثال عنصر .S→XYZ نشان می‌دهد که XYZ یک دستگیره است که می‌توان آن را به S کاهش داد.

عنصر انتقالی: عنصری که کاهشی نباشد. به عنوان مثال S→X.YZ یک عنصر انتقالی است. هر عنصر انتقالی نشان دهنده فرضیه‌ای در مورد دستگیره بعدی است به عنوان مثال S→X.YZ نشان می‌دهد که دستگیره ممکن است از Y به دست آید.

نماد بلافاصله سمت راست علامت نقطه باعث پیشرفت علامت نقطه و یافتن دستگیره می‌شود. عنصر انتقالی S→α.aβ را در نظر می‌گیریم به طوریکه a پایانه است. عنصر S→α.aβ نشان می‌دهد که اگر a در نماد جاری ورودی باشد، a از ورودی حذف و S→α.aβ به S→αa.β تبدیل می‌گردد. S→α.Xβ را در نظر می‌گیریم به طوریکه X غیرپایانه است. S→α.Xβ این فرضیه را بیان می‌کند که دستگیره از X باید به دست آید با کاهش X علامت نقطه از X عبور می‌کند و عنصر S→α.Xβ به S→αX.β تبدیل می‌گردد.

با استفاده از عناصر LR و گروهبندی آنها می‌توان یک ماشین متناهی قطعی ایجاد کرد. هر یک از این گروهها را به عنوان یک حالت در جدول تجزیه استفاده می‌کنیم. به منظور تشریح الگوریتم گرامر ذیل را در نظر می‌گیریم.

E→E+T
E→T
T→id

ابتدا قاعده تولید S→E را به گرامر اضافه می‌کنیم. گرامر جدید به صورت است.

S → E
E→E+T
E→T
T→id

اولین عنصر S→.E است که فرضیه ای را مورد دستگیره بیان می‌کند. علامت نقطه قبل از E نشان می‌دهد که ممکن است دستگیره از E به دست آید. با توجه به قواعد تولید E→E+T و

E→T برای تولید E باید E+T یا T یافت شود تا بتوان آنها را به E کاهش داد، در نتیجه ممکن است دستگیره بعدی T و یا E+T باشد. این مطلب را با عناصر E→.E+T و E→.T نشان می‌دهیم. در نتیجه عناصر E→.E+T و E→.T را به مجموعه عناصر موجود اضافه می‌گردد این دو عنصر دو فرضیه یا امکان جدید را جهت یافتن دستگیره بعدی نشان می‌دهند. مجموعه فرضیه‌ها یا عناصر به شرح ذیل هستند.

S→.E
E→.E+T
E→.T

با بررسی این عناصر LR (فرضیه‌ها)، فرضیه دیگری با استفاده از E→.T مطرح می‌گردد با توجه به قاعده تولید T→id ،ممکن است دستگیره از T به دست آید. این فرضیه با استفاده از عنصر T→.id نشان داده می‌شود. مجموعه فرضیه‌های موجود به صورت ذیل هستند.

S→.E
E→.E+T
E→.T
T→.id

با بررسی بیشتر این فرضیات یا عناصر، فرضیه جدیدی قابل استنتاج نیست. این مجموعه راS0 می‌نامیم.

مجموعه عناصر به دست آمده از عنصر S→.E را closure({ S→.E}) می‌نامیم. closure(I) مجموعه تمام فرضیاتی است که از I قابل استنتاج است. برای محاسبه closure(I) از دو قانون ذیل تا زمانیکه استفاده از آنها عنصر جدیدی را به مجموعه اضافه می‌کند می‌توان استفاده کرد.

۱- I به closure(I) اضافه می‌گردد.

۲- اگر A→X.YZ در closure(I) و قاعده تولید Y→.αβ در گرامر باشد آنگاه Y→.αβ به closure(I) اضافه می‌گردد.

مثال۳-۵۷ با توجه به گرامر زیر closure({S→.E}) را محاسبه کنید.

S → E
E→E+T
E→T
T→id

با استفاده از قانون ۱ عنصر S→.E به closure({S→.E}) اضافه می‌گردد. در نتیجه:

closure({S→.E})= {S→.E}

با استفاده از قانون ۲، S→.E در closure({S→.E}) و E→E+T وE→T در گرامر است، در نتیجه E→.E+T و E→.T به closure({S→.E}) اضافه می‌گردد، در نتیجه:

closure({S→.E})={S→.E, E→.E+T , E→.T }

با استفاده از قانون ۲، E→.T در closure({S→.E}) و T→id در گرامر است در نتیجه T→.id نیز به مجموعه closure({S→.E}) اضافه می‌گردد. در نتیجه:

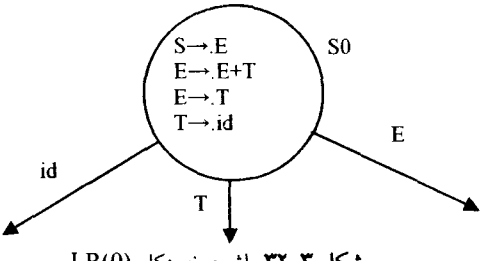
closure({S→.E})={ S→.E, E→.E+T,E→.T, T→.id}

اعمال قوانین ۱و ۲ در closure({S→.E}) عنصر جدیدی را اضافه نمی‌کند. در نتیجه:

closure({S→.E})={ S→.E, E→.E+T,E→.T, T→.id}

مجموعه فرضیات قابل استنتاج از یک عنصر را از طریق قوانین محاسبه closure و یا از طریق مراحل قدم به قدم که در ابتدا ذکر شد، می‌توان به دست آورد. به منظور درک بهتر محاسبه مجموعه عناصر قابل استنتاج، در ادامه از هر دو روش، مجموعه عناصر قابل استنتاج را محاسبه می‌کنیم ولی برای ساخت جدول تجزیه نیازی به استفاده از هر دو روش نیست.

با توجه به عناصر LR موجود در S0، سمت راست علامت نقطه علائم، E,T و id است. در نتیجه، علامت نقطه فقط به ازای id, T,E قادر به پیشروی است. قسمت اول ماشین خودکار قطعی در شکل ۳-۳۲ نشان داده شده است.



شکل ۳-۳۲ ماشین خودکار LR(0)

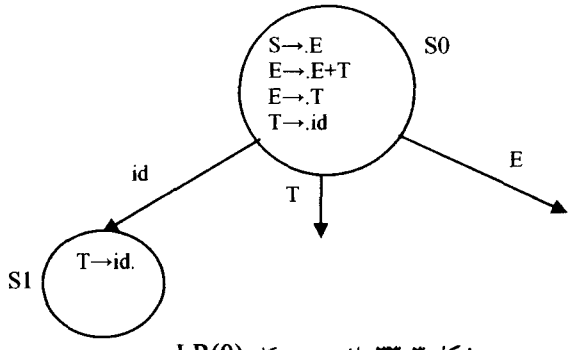
در ذیل به بررسی تاثیر id,E و T روی مجموعه S0 می‌پردازیم.

نماد id باعث می‌شود علامت نقطه در عنصر T→.id، پیشروی کرده و از id عبور کند. در این صورت عنصر T→.id به T→id. تبدیل می‌شود. این عنصر یک عنصر کاهشی است و عنصر کاهشی نشان دهنده یافتن یک دستگیره است. عنصر T→id. نشان می‌دهد که id دستگیره است. سمت راست نقطه در T→id. نماد دیگری وجود ندارد بنابراین فرضیه جدیدی قابل استنتاج نیست در نتیجه مجموعه عناصر قابل استنتاج از T→id ، فقط شامل T→id. است، این مجموعه را S1 می‌نامیم.

مجموعه S1 با استفاده از قوانین محاسبه closure({T→id}) نیز قابل محاسبه است. با استفاده از قوانین محاسبه closure نتیجه ذیل به دست می‌آید.

closure({T→id.})= {T→id.}

در نتیجه id باعث تغییر حالت از S0 به S1 می‌شود. این تغییر حالت در شکل ۳-۳۳ نشان داده شده است.



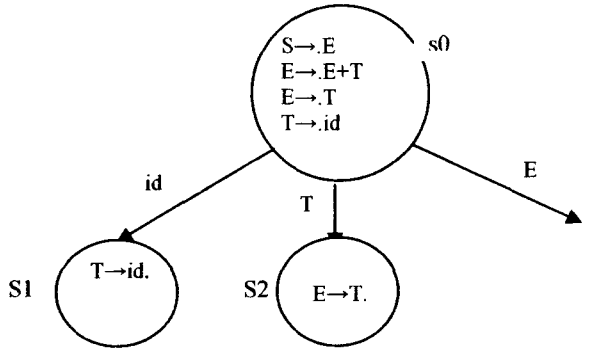
شکل ۳-۳۳ ماشین خودکار LR(0)

نماد T باعث می‌گردد از مجموعه عناصر موجود در S0 عنصر E→.T به عنصر کاهشی E→T. تبدیل گردد. با توجه به اینکه نقطه به انتهای سمت راست رسیده است، فرضیه جدیدی قابل استنتاج نیست.

این نتیجه با استفاده از قوانین محاسبه closure نیز قابل اثبات است. استفاده از این قوانین نتیجه ذیل را به دست می‌دهد.

closure({E→T.})= {E→T.}

مجموعه {E→T.} را S2 می‌نامیم. در نتیجه T باعث تغییر حالت از S0 به S2 می‌شود. این تغییر حالت در شکل ۳-۳٤ نشان داده شده است.

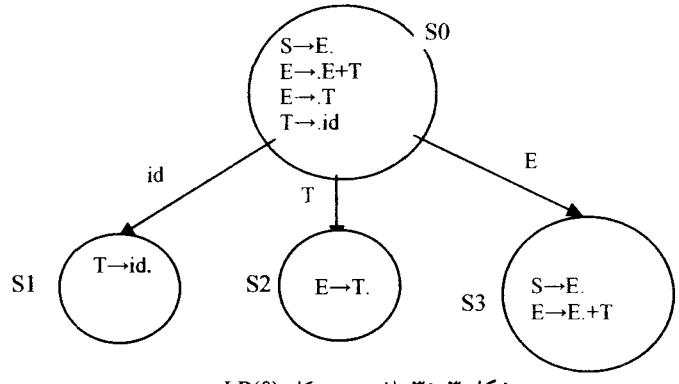


شکل ۳-۳٤ ماشین خودکار LR(0)

نماد E باعث می‌گردد از مجموعه عناصر موجود در S0 عناصر E→.E+T و S→.E تحت تاثیر قرار گیرد. عنصر E→.E+T به E→E.+T و عنصر S→.E به S→E. تبدیل می‌گردد. با استفاده از عناصر E→E.+T و S→E. فرضیه جدیدی قابل استنتاج نیست.

اگر قوانین محاسبهclosure روی مجموعه {S→E. ,E→E.+T} اعمال کنیم عنصر جدیدی را اضافه نمی‌کند.

{E→E.+T,S→E.} را S3 می‌نامیم. در نتیجه E باعث تغییر حالت از S0 به S3 می‌شود. نتیجه در شکل ۳-۳۵ نشان داده شده است.



شکل ۳-۳۵ ماشین خودکار LR(0)

همانطور که ملاحظه می‌گردد آنچه به دست می‌آید شبیه DFA است. از مجموعه حالات جدید S1,S2,S3 فقط در عناصر S3 نقطه امکان حرکت و طرح فرضیات جدید را دارد. مجموعه‌های S1 و S2 فقط دارای عنصر کاهشی بوده و امکان پیشرفت علامت نقطه در آنها وجود ندارد. در نتیجه فقط S3 را بررسی می‌کنیم.

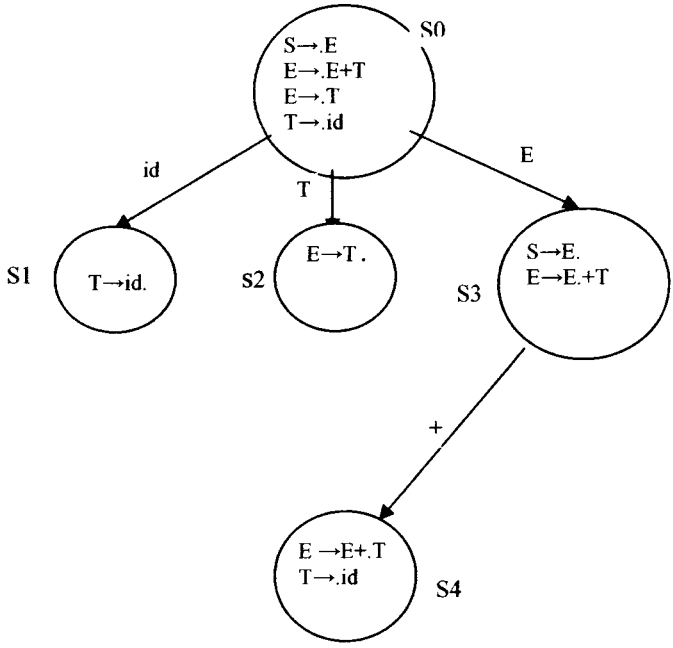
نماد + باعث می‌گردد عنصر E→E.+T، به عنصر E→E+.T تبدیل گردد. با استفاده از عنصر E→E+.T، فرضیه یافتن یک دستگیره از T ارائه می‌شود. با توجه به قاعده تولید T→id عنصر T→.id به مجموعه عناصر اضافه می‌شود. مجموعه فرضیه‌های جدید به صورت ذیل است.

E→E+.T
T→.id

اگر قوانین محاسبه closure را روی E→E+.T اعمال کنیم مجموعه ذیل به دست می‌آید.

{E→E+.T, T→.id }

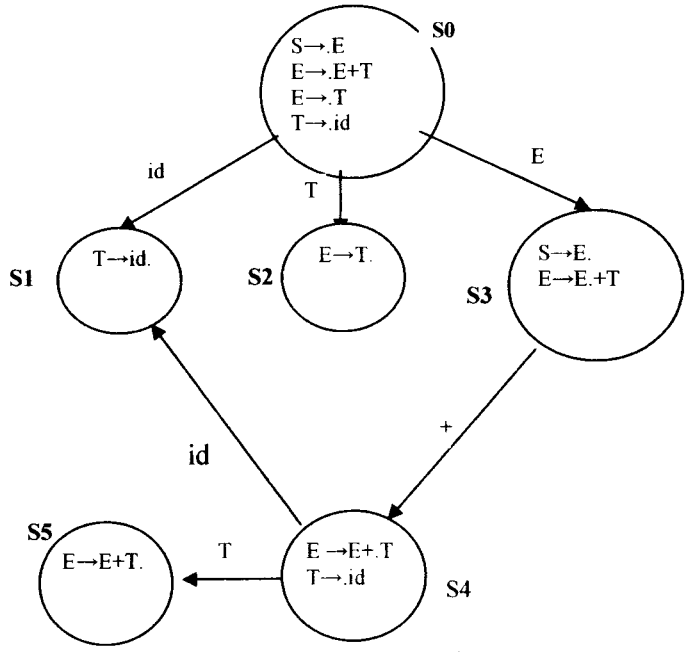
مجموعه عناصر {E→E+.T,T→.id} را S4 می‌نامیم. در نتیجه + باعث تغییر حالت از S3 به S4 می‌شود. نتیجه بررسی S3 در شکل ذیل نشان داده شده است.



شکل ۳-۳۶ ماشین خودکار LR(0)

در دو عنصر E→E+.T ،T→.id از حالت S4 امکان حرکت و پیشروی نقطه وجود دارد. نمادهای id و T سمت راست نقطه قرار دارند، بنابراین این علائم باعث حرکت نقطه می‌شوند. T باعث می‌شود عنصر E→E+.T به عنصر کاهشی .E→E+T تبدیل شود. نقطه به انتهای سمت راست رسیده است در نتیجه امکان استنتاج فرضیات جدید وجود ندارد. مجموعه {E →E+T.} را S5 می‌نامیم. در نتیجه T باعث تغییر حالت از S4 به S5 می‌گردد.

id باعث می‌شود عنصر T→.id به عنصر T→id. تبدیل شود. با استفاده از T→.id ، عنصر جدیدی قابل استنتاج نیست. مجموعه {T→id.} همان مجموعه S1 است که در مراحل قبلی تولید گردیده است. در نتیجه id باعث تغییر حالت از S4 به S1 می‌گردد. نتیجه بررسی S4 در شکل ذیل نشان داده شده است.



شکل ۳-۳۷ ماشین خودکار LR(0)

در عناصر مجموعه S5 امکان پیشرفت نقطه وجود ندارد. در نتیجه حالت جدیدی به ماشین خودکار اضافه نخواهد شد. ماشین خودکار قطعی به دست آمده شبیه DFA است. با استفاده از این ماشین خودکار می‌توان جدول تجزیه را استخراج کرد.

می‌توان ماشین خودکار را با استفاده از جدول نیز نمایش داد و جدول تجزیه را از آن تولید کرد. ساختار کلی جدول تجزیه در ذیل نشان داده شده است.

جدول ۳-۲۵ ساختار جدول تجزیه

حالات نمودار تغییر حالت	action		goto
	پایانه‌های گرامر	$	غیر پایانه‌های گرامر

در این جدول منظور از حالات، حالتهای ماشین خودکار است، به منظور اختصار فقط شماره حالت را درج می‌کنیم، به عنوان مثال به جای S0 در جدول 0 درج می‌کنیم. به عنوان مثال ماشین خودکار گرامر مورد نظر شش حالت از S0 تا S5 دارد در نتیجه جدول تجزیه نیز شش سطر دارد. بخش goto شامل غیر پایانه‌ها و بخش action شامل پایانه‌های گرامر و علامت $ است. جدول تجزیه گرامر مورد نظر به صورت ذیل است.

جدول ۳-۲۶ ساختار جدول تجزیه

حالات	action			goto	
	id	+	$	E	T
0					
1					
2					
3					
4					
5					

برای سهولت پر کردن جدول به هر قاعده تولید یک شماره اختصاص می‌دهیم.

1- S→E
2- E→E+T
3- E→T
4- T→id

نحوه پر کردن جدول تجزیه را با استفاده از ماشین خودکار و رشته ورودی تشریح می‌کنیم. در ذیل هر سطر(حالت) جدول را بررسی کرده و تکمیل می‌کنیم.

حالت S0: با توجه به ماشین خودکار، همه عناصر در حالت صفر انتقالی هستند. نمادهای E,T و id که بلافاصله بعد از علامت نقطه قرار دارند باعث تغییر حالت از حالت S0 می‌شوند. در ذیل به بررسی تاثیر هریک از این نمادها می‌پردازیم.

با توجه به ماشین خودکار، نماد id باعث تغییر حالت از حالت S0 به S1 می‌شود، و عنصر انتقالی، T→.id به عنصر .T→id تبدیل می‌گردد. این عمل همراه با انتقال id به پشته است. در نتیجه action[0,id]=shift 1 است. shift به معنی انتقال id به پشته است و عدد 1 نشان می‌دهد بعد از این انتقال، تجزیه کننده به حالت S1 می‌رود. به منظور اختصار به جای shift 1 از S1 استفاده می‌کنیم در نتیجه:

action[0,id]=S1

نمادهای + و $ در حالت S0 بی‌تاثیر هستند زیرا با نمادهای + و$ از S0 خروجی وجود ندارد. بدین جهت اگر تجزیه کننده به این حالات برسد خطا رخ داده است. در نتیجه:

action [0,+]=error
action[0,$]=error

با توجه به ماشین خودکار، نماد T باعث تغییر حالت، از حالت S0 به S2 می‌گردد و عنصر انتقالی، E→.T به عنصر E→T. تبدیل می‌گردد. با توجه به اینکه T یک غیر پایانه است در نتیجه روی ورودی بی تاثیر است و فقط باعث تغییر حالت می‌گردد. به همین دلیل در قسمت goto جدول موثر است. با توجه به ماشین خودکار نتیجه ذیل به دست می‌آید.

goto[0,T]=2

با توجه به ماشین خودکار، غیر پایانه E باعث انتقال از حالت S0 به S3 می‌گردد و عنصر انتقالی، T→.E+T به عنصر T→E.+T و عنصر S→.E به S→E. تبدیل می‌گردد. با توجه به اینکه E یک غیر پایانه است در نتیجه روی ورودی بی تاثیر است و فقط باعث تغییر حالت می‌گردد. در نتیجه:

goto[0,E]=3

نتیجه بررسی حالت S0 در جدول ذیل نشان داده شده است.

جدول ۳-۲۷ جدول تجزیه LR(0)

حالت	action			goto	
	id	+	$	T	E
0	S1	error	error	2	3
1					
2					
3					
4					
5					

حالت S1: این حالت شامل عنصر کاهشی T→id. است. در این حالت یک دستگیره یافت شده است. در نتیجه در این حالت با توجه به قاعده تولید T→id کاهش انجام می‌شود. بنابراین در سطر 1 جدول reduce T→id قرار می‌گیرد. به منظور اختصار از حرف r برای reduce و از شماره قاعده تولید در گرامر به جای T→id استفاده می‌کنیم. در نتیجه به جای reduce T→id از r4 استفاده می‌کنیم. چون عمل کاهش صرف نظر از نماد ورودی انجام می‌شود، در نتیجه برای همه پایانه‌ها r4 را درج می‌کنیم. در نتیجه :

action[1,id]=r4

action[1,+]=r4
action[1,$]=r4

نتیجه بررسی S1 در ذیل ارائه شده است.

جدول ۳-۲۸ جدول تجزیه LR(0)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	r4	r4	r4		
2					
3					
4					
5					

حالت S2: حالت S2 شامل عنصر کاهشی .E→T است. در این حالت یک دستگیره یافت شده است. درحالت S2 با توجه به قاعده تولید E→T کاهش انجام می‌شود. در نتیجه در سطر 2 جدول باید reduce E→T قرار گیرد. به منظور اختصار به جای reduce E→T از r3 استفاده می‌کنیم. چون عمل کاهش صرف نظر از نماد ورودی انجام می‌شود، در نتیجه برای همه پایانه‌ها r3 را درج می‌کنیم. در نتیجه :

action[2,id]=r3
action[2,+]=r3
action[2,$]=r3

نتیجه این مرحله در جدول ذیل ارائه شده است.

جدول ۳-۲۹ جدول تجزیه LR(0)

حالات	action			Goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	r4	r4	r4		
2	r3	r3	r3		
3					
4					
5					

حالت S3: حالت S3 شامل عنصر کاهشی قاعده تولید اضافه شده .S→E است در نتیجه، نماد شروع تولید شده است، اگر نماد ورودی $ باشد، رشته ورودی نیز تمام شده است (زیرا$ نشان دهنده پایان رشته ورودی است)، در نتیجه رشته ورودی پذیرفته می‌گردد. در نتیجه:

action[3,$]= accept

با توجه به ماشین خودکار، نماد + باعث تغییر حالت از حالت S3 به S4 می‌گردد و عنصر انتقالی، E→E.+T به عنصر E→E+.T تبدیل می‌گردد. این عمل همراه با انتقال + به پشته است. در نتیجه :

action[3,+]=S4

نماد id در حالت S3 بی تاثیر است، در نتیجه اگر در حالت S3 نماد ورودی id باشد خطا رخ می‌دهد، در نتیجه:

action[3,id]=error

جدول ۳- ۳۰ جدول تجزیه LR(0)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	r4	r4	r4		
2	r3	r3	r3		
3	error	s4	accept		
4					
5					

حالت S4: با توجه به ماشین خودکار، همه عناصر در حالت S4 انتقالی هستند، در نتیجه نمادهای T و id که بلافاصله بعد از علامت نقطه قرار دارند باعث تغییر حالت ازS4 می‌شوند. در ذیل به بررسی تاثیر هرکدام از این نمادها می‌پردازیم.

با توجه به ماشین خودکار، نماد id باعث تغییر حالت از حالت S4 به S1 می‌گردد. عنصر انتقالی T→.id به عنصرکاهشی .T→id تبدیل می‌گردد. این عمل همراه با انتقال id به پشته است. در نتیجه: action[4,id]=shift 1 است. در نتیجه:

action[4,id]=S1

نمادهای + و $ در این حالت بی تاثیر هستند بدین جهت اگر تجزیه کننده به این حالات برسد خطا رخ داده است. در نتیجه:

action [4,+]=error
action[4,$]=error

با توجه به نمودار تغییر حالت، نماد T باعث تغییر حالت از حالت S4 به حالت S5 می‌گردد و عنصر انتقالی، E→E+.T به عنصر E→E+T. تبدیل می‌گردد. با توجه به اینکه T یک غیر پایانه است در نتیجه روی ورودی بی تاثیر است و فقط باعث تغییر حالت می‌گردد. در نتیجه goto[4,T]=5 است.

جدول ۳-۳۱ جدول تجزیه LR(0)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	r4	r4	r4		
2	r3	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5					

حالت S5: این حالت شامل عنصر کاهشی E→E+T. است. در این حالت یک دستگیره یافت شده است. در نتیجه در این حالت با توجه به قاعده تولید E→E+T کاهش انجام می‌شود. در سطر 5 جدول reduce E→E+T قرار می‌گیرد. به منظور اختصار به جای reduce E→T از r2 استفاده می‌کنیم. چون عمل کاهش صرف نظر از نماد ورودی انجام می‌شود، در نتیجه برای همه پایانه‌ها r2 را درج می‌کنیم. در نتیجه :

action[5,+]=r2
action[5,$]=r2
action[5,id]=r2

خانه‌های خالی را با error پر می‌کنیم. نتیجه بررسی S5 در جدول ذیل ارائه شده است.

جدول ۳-۳۲ جدول تجزیه LR(0)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	r4	r4	r4		
2	r3	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5	r2	r2	r2		

روش ساخت جدول تجزیه LR(0) را می‌توان به صورت ذیل خلاصه کرد.

- اگر در ماشین خودکار پایانه a باعث تغییر حالت از Si به Sj شود. action[i,a]=shift j یا به اختصار action[i,a]=sj را در قسمت action قرار می‌دهیم. Shift j به معنی انتقال پایانه a و شماره حالت j به پشته است. به عنوان مثال id باعث تغییر حالت از S0 به S1 می‌گردد. این تغییر حالت را با action[0,id]=S1 نشان می‌دهیم. دقت کنید این روش فقط برای پایانه‌ها است.

- اگر در ماشین خودکار غیر پایانه X باعث تغییر حالت از Si به Sj شود، goto[i,X]=j را در قسمت goto قرار می‌دهیم. به عنوان مثال غیر پایانه E باعث تغییر حالت از S0 به S3 می‌شود. این تغییر حالت را با goto[0,E]= 3 نشان می‌دهیم. دقت کنید این روش فقط برای غیرپایانه‌ها است.

- اگر در ماشین خودکار، حالت Si شامل عنصر کاهشی .A→α باشد (به جز عنصر کاهشی حاصل از قاعده تولید اضافه شده مانند .S→E)، در جدول تجزیه برای هر پایانه a، action[i,a]= reduce A→α یا برای اختصار action[i,a]= rn را قرار می‌دهیم (rn به معنی کاهش با A→α است، r اختصار کلمه reduce و n شماره قاعده تولید A→α در گرامر است.) زیرا عنصر کاهشی نشان دهنده یافتن دستگیره است و در نتیجه می‌توان کاهش را انجام داد. به عنوان مثال S1 شامل عنصر کاهشی .T→id است در نتیجه:

action[1,id]= reduce T→id
action[1,+] =reduce T→id
action[1,$] =reduce T→id

برای اختصار از عبارات ذیل استفاده می‌کنیم.

action[1,id]= r4
action[1,+] = r4
action[1,$] = r4

- اگر در ماشین خودکار حالت Si شامل عنصر کاهشی قاعده تولید اضافه شده باشد،(مانند .S→E ، به طوریکه E نماد شروع گرامر باشد) و نماد جاری $ باشد، action[i,$]=accept قرار می‌دهیم. اگر تجزیه کننده به این حالت برسد رشته پذیرفته می‌شود. زیرا کاهش S→E نماد شروع گرامر را تولید می‌کند، و نماد $ نشان دهنده پایان رشته ورودی است، در نتیجه تجزیه با موفقیت تمام شده است.

به عنوان مثال حالت S3 شامل عنصر کاهشی قاعده تولید اضافه شده .S→E است، با کاهش این عنصر، نماد شروع گرامر ایجاد شده است، اگر در این وضعیت نماد جاری $ باشد، رشته ورودی نیز تمام شده است در نتیجه رشته پذیرفته می‌شود. به همین دلیل action[3,$]=accept قرار می‌دهیم.

- مکانهای خالی جدول را error قرار می‌دهیم اگر تجزیه کننده به این حالات برسد، رشته ورودی رد می‌شود. در برخی موارد برای اختصار مکانهای error را خالی قرار می‌دهیم.
- حالت که شامل عنصر قاعده تولید اضافه شده است حالت شروع ماشین خودکار است.

جدول ۳-۳۳ جدول تجزیه LR(0)

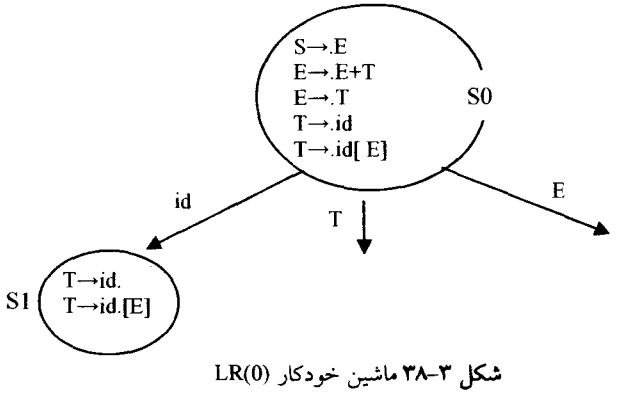
حالات	action			goto	
	id	**+**	**$**	**T**	**E**
0	s1	error	error	2	3
1	r4	r4	r4		
2	r3	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5	r2	r2	r2		

جدول فوق را جدول LR(0) می‌نامیم، اگر تجزیه کننده، از این جدول برای تجزیه استفاده کند ، تجزیه کننده را LR(0) می‌نامیم و گرامری که بتوان برای آن تجزیه کننده LR(0) ساخت، گرامر LR(0) می‌نامیم. به عبارت دیگر گرامر LR(0) گرامری است که می‌توان برای آن تجزیه کننده LR(0) ایجاد کرد. روش LR(0) قادر به تولید جدول تجزیه برای همه گرامرها نیست.

مثال ۳-۵۸ گرامر ذیل را در نظر می‌گیریم.

S→E
E→E+T
E→T
T→id
T→id [E]

. در شکل ۳-۳۷ بخشی از ماشین خودکار مربوط به گرامر نشان داده شده است.



شکل ۳-۳۸ ماشین خودکار LR(0)

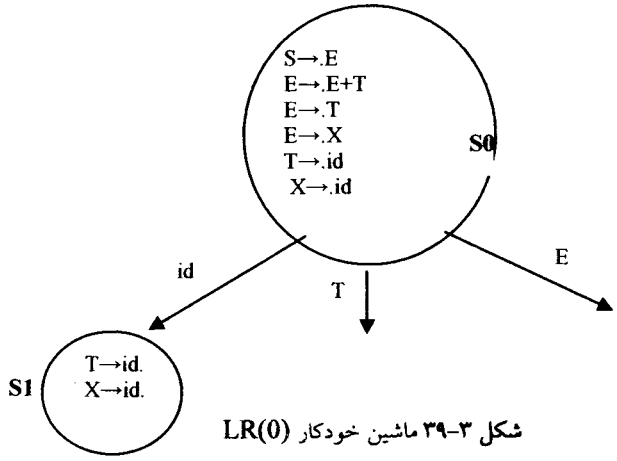
با توجه به ماشین خودکار شکل ۳-۳۸، در حالت S1 هم عنصر انتقالی و هم عنصر کاهشی وجود دارد یعنی هم امکان کاهش و هم امکان انتقال وجود دارد. در نتیجه مشخص نیست

در این حالت باید عمل کاهش انجام شود یا انتقال. این وضعیت را برخورد انتقال/کاهش[1] می‌نامیم. اگر در گرامری برخورد انتقال/ کاهش رخ دهد، گرامر LR(0) نیست.

مثال۳-۵۹ گرامر ذیل را در نظر می‌گیریم.

S→E
E→E+T
E→T
E→X
T→id
X→ id

با توجه به گرامر بخشی از ماشین خودکار در شکل ۳-۳۹ ارائه شده است.



شکل ۳-۳۹ ماشین خودکار LR(0)

با توجه به شکل ۳-۳۹ در حالت S1 دو عنصر کاهشی وجود دارد یعنی دو امکان برای کاهش وجود دارد. ولی مشخص نیست در حالت S1 کدام کاهش باید انجام شود. این وضعیت را برخورد کاهش/ کاهش[2] می‌نامیم. با توجه به این برخورد گرامر مورد نظر LR(0) نیست.

اگر در گرامری برخورد انتقال/کاهش یا کاهش/کاهش رخ دهد، گرامرLR(0) نیست. با دقت در مثالهای ذکر شده مشخص می‌شود برای به وجود آمدن برخورد انتقال/کاهش یا کاهش/کاهش، باید حداقل یک عنصر کاهشی وجود داشته باشد. به همین دلیل برای تست LR(0) بودن گرامر، ابتدا ماشین خودکار را رسم می‌کنیم، سپس حالاتی که شامل عنصر کاهشی هستند را بررسی می‌کنیم. اگر حالتی یافت شود که دارای دو عنصر کاهشی است و

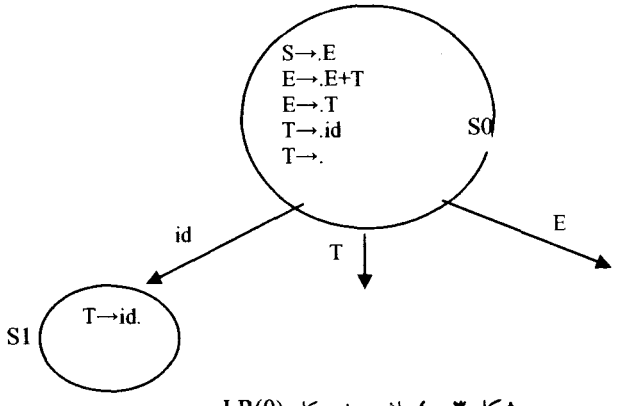
1. Shift/Reduce Conflict
1. Reduce/Reduce Conflict

یا عنصر کاهشی و عنصر انتقالی است گرامر LR(0) نیست. در این تست عنصر کاهشی حاصل از قاعده تولید اضافه شده مستثنی است. به عنوان نمونه عنصر .S→E در این تست در نظر گرفته نمی‌شود.

مثال۳-۶۰ گرامر ذیل را در نظر می‌گیریم.

S→E
E→E+T
E→T
T→∈
T→id

ماشین خودکار LR(0) گرامر را رسم می کنیم.



شکل ۳-۴۰ ماشین خودکار LR(0)

با توجه به ماشین خودکار، در حالت S0 هم عنصر انتقال و هم عنصر کاهشی وجود دارد، بنابراین امکان کاهش و انتقال وجود دارد. مشخص نیست در این حالت باید عمل کاهش انجام شود یا انتقال. در نتیجه برخورد انتقال/کاهش رخ می‌دهد. با توجه به این برخورد، گرامر مورد نظر LR(0) نیست.

۳-۱۸-۲- روش SLR(1)

روش LR(0) که در بخش قبلی مورد بررسی قرار گرفت، بسیار ضعیف است، به طوریکه در مورد بسیاری از گرامرها قابل استفاده نیست. علت ضعف LR(0) عدم توجه به نماد جاری رشته ورودی در هنگام کاهش یک عنصر کاهشی است. به عبارت دیگر LR(0) ، با تشخیص یک عنصر کاهشی، بدون توجه به ورودی برای همه پایانه‌ها اقدام به کاهش می‌کند، روش SLR(1) هوشمندانه‌تر از LR(0) است، روش SLR(1) عنصر کاهشی .N→α را وقتی به

Nکاهش می‌دهد که نماد جاری ورودی در follow(N) باشد. زیرا طبق تعریف follow(N) نمادهایی هستند که بلافاصله بعد ازN قرار می‌گیرند، اگر a نماد ورودی باشد و در follow(N) نباشد، کاهش .N→α باعث تولید شبه جمله Na می‌گردد، بنابراین a بلافاصله بعد از N قرار می‌گیرد، در نتیجه پایانه a عضو follow(N) است (زیرا a بلافاصله بعد از N قرار گرفته است)، که با فرض اول کهa در follow(N) نیست در تناقض است. در نتیجه، کاهش عنصر کاهشی .N→α فقط برای پایانه‌های follow(N) قابل انجام است.

در روش SLR(1) مانند روش LR(0) ابتدا ماشین خودکار را رسم می‌کنیم، سپس با استفاده از آن جدول تجزیه را ایجاد می‌کنیم. تفاوت SLR(1) و LR(0) در نحوه تولید جدول از ماشین خودکار است. اگر به جدول LR(0) دقت کنید ملاحظه می‌گردد، در حالتی که عنصر کاهشی وجود دارد صرفنظر از نماد ورودی عمل کاهش انجام می‌شود. در حالیکه در جدول SLR(1) فقط برای پایانه‌های follow(N) کاهش انجام می‌گیرد. جدول تجزیه SLR(1) این گرامر در ذیل نشان داده شده است.

جدول ۳-۳۴ جدول تجزیه SLR(1)

حالات نمودار تغییر حالت	action		goto
	پایانه‌های گرامر	$	غیر پایانه‌های گرامر

در این جدول منظور از حالات، حالتهای ماشین خودکار است. در نتیجه LR(0) و SLR(1) تعداد حالات یکسانی تولید می‌کنند. بخش goto شامل غیر پایانه‌ها و بخش action شامل پایانه‌های گرامر و علامت $ است. نحوه پر کردن این جدول را با استفاده از گرامر ذیل تشریح می‌کنیم.

E→E+T
E→T
T→id

در ابتدا به گرامر یک قاعده تولید به صورت ذیل اضافه می‌کنیم.

غیر پایانه شروع گرامر → یک غیر پایانه

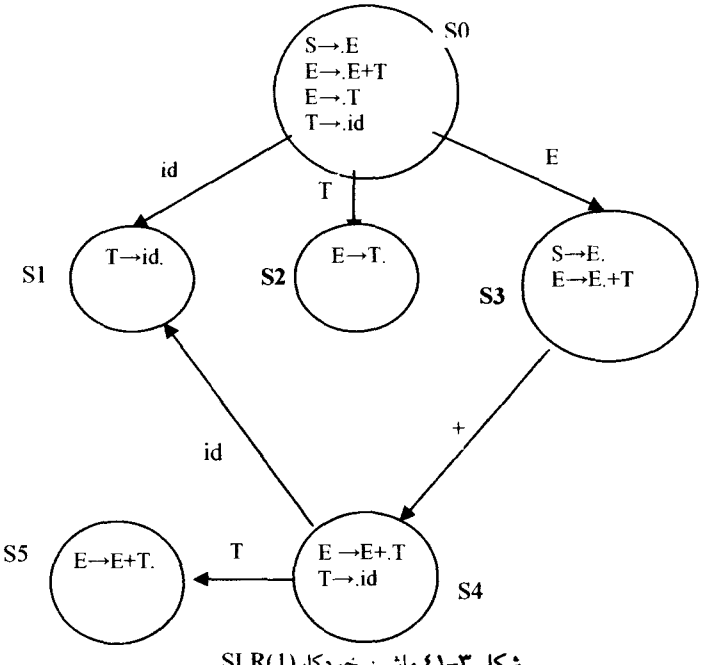
قاعده تولید S→E را به گرامر مورد نظر اضافه می‌کنیم. مانند LR(0) برای هر قاعده تولید یک شماره در نظر می‌گیریم، گرامر مورد نظر به صورت ذیل خواهد شد.

1- S → E

2- E→E+T
3- E→T
4- T→id

با توجه به گرامر، هرگاه تجزیه کننده موفق به کاهش S→E گردد و نماد ورودی نیز $ باشد، تجزیه با موفقیت به پایان می‌رسد.

با توجه به گرامر، پایانه‌های id و + از گرامر و $ به بخش action و غیر پایانه‌های E و T به بخش goto اضافه می‌گردند. با استفاده از روشی که در LR(0) ذکر شد، ماشین خودکار را رسم می‌کنیم. ماشین خودکار گرامر در شکل ذیل ارائه شده است.



شکل ۳-۴۱ ماشین خودکار SLR(1)

حالات ماشین خودکار، S0,S1,S2,S3,S4,S5 در قسمت حالات اضافه می‌گردند. برای اختصار از حرف S صرفنظر کرده و فقط شماره آن را در جدول درج می‌کنیم. سطرهای جدول را به کمک ماشین خودکار و گرامر تکمیل می‌کنیم.

جدول۳-۳۵ جدول تجزیه SLR(1)

حالات	action			goto	
	id	+	$	E	T
0					
1					
2					
3					
4					
5					

در ذیل هر سطر جدول را به وسیله ماشین خودکار تکمیل می‌کنیم.

حالت S0: با توجه به ماشین خودکار، همه عناصر در حالت S0 انتقالی هستند در نتیجه این عناصر با دریافت نمادها باعث تغییر حالت می‌شوند. نمادهای E,T و id که بلافاصله بعد از علامت نقطه قرار دارند باعث تغییر حالت از حالت S0 می‌شوند. در ذیل به بررسی تاثیر هریک از این نمادها می‌پردازیم.

با توجه به ماشین خودکار، نماد id باعث تغییر حالت از حالت S0 به S1 می‌گردد. و عنصر انتقالی ، T→.id به عنصر T→id. تبدیل گردد. این عمل همراه با انتقال id به پشته است. در نتیجه action[0,id]=shift 1 است. shift به معنی انتقال id به پشته است و عدد1 نشان می‌دهد بعد از این انتقال نمودار به حالت یک می‌رود. به منظور اختصار به جای shift 1 از S1 استفاده می‌کنیم، در نتیجه:

action[0,id]=S1

نمادهای + و $ روی این حالت بی تاثیر هستند یعنی با نمادهای + و $ از حالت S0 خروجی وجود ندارد. بدین جهت اگر تجزیه کننده به این حالات برسد خطا رخ داده است. در نتیجه:

action [0,+]=error
action[0,$]=error

با توجه به ماشین خودکار، نماد T باعث تغییر حالت از S0 به S2 می‌گردد، و عنصر انتقالی، E→.T به عنصر E→T. تبدیل می‌گردد. با توجه به اینکه T یک غیر پایانه است در نتیجه روی ورودی بی تاثیر است و فقط باعث تغییر حالت می‌گردد. به همین دلیل در قسمت goto جدول موثر است. در نتیجه با توجه به نمودار:

goto[0,T]=2

با توجه به ماشین خودکار، نماد E باعث انتقال از حالت S0 به S3 می‌گردد و عنصر انتقالی، T→.E+T به T→E.+T و عنصر S→.E به S→E. تبدیل می‌گردد. با توجه به اینکهE یک غیر

پایانه است در نتیجه روی ورودی بی تاثیر است و فقط باعث تغییر حالت می‌گردد. در نتیجه goto[0,E]=3 است. نتیجه بررسی حالت S0 در جدول ذیل نشان داده شده است.

جدول ۳-۳۶ جدول تجزیه SLR(1)

حالات	action			goto	
	id	+	$	T	E
0	S1	error	error	2	3
1					
2					
3					
4					
5					

حالت S1: این حالت شامل عنصر کاهشی .T→id است. در این حالت یک دستگیره یافت شده است. اما این سوال مطرح می‌شود که در چه شرائطی کاهش انجام می‌شود؟ پاسخ آن است که اگر نماد جاری ورودی بتواند بلافاصله بعد از T قرار گیرد انجام کاهش بلامانع است. نمادهای که می‌توانند بلافاصله بعد ازT قرار گیرند توسط Follow(T) محاسبه می‌گردد. در نتیجه برای پایانه‌های Follow(T) انجام کاهش بلامانع است.

Follow(T)={+,$}

در نتیجه action[1,+]=reduce T→id و action[1,$]=reduce T→id است. به منظور اختصار از action[1,+]=r4 و action[1,$]=r4 استفاده می‌کنیم. زیرا T→id قاعده تولید شماره ٤ در گرامر است. اگر نماد جاری id باشد خطا رخ می‌دهد زیرا id نمی‌تواند بعد از T قرار گیرد یا به عبارت دیگر id در follow(T) قرار ندارد، در نتیجه action[1,id]=error .

جدول ۳-۳۷ جدول تجزیه SLR(1)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	error	r4	r4		
2					
3					
4					
5					

حالت S2: این حالت شامل عنصر کاهشی .E→T است. در این حالت یک دستگیره یافت شده است. اما این سوال مطرح می‌شود که در چه شرائطی باید عمل کاهش انجام شود؟ پاسخ آن است که اگر نماد جاری ورودی بتواند بلافاصله بعدE قرار گیرد انجام کاهش بلامانع است. نمادهای که می‌توانند بلافاصله بعد ازE قرار گیرند توسط Follow(E) محاسبه می‌گردد. در نتیجه برای پایانه‌های follow(E) انجام کاهش بلامانع است.

follow(E)={+,$}

در نتیجه action[2,+]=reduce E→T و action[2,$]=reduce E→T است. به منظور اختصار در جدول این موارد به صورت action[2,+]=r3 و action[2,$]=r3 نشان داده شده است، اگر نماد جاری id باشد خطا رخ می‌دهد در نتیجه action[2,id]=error .

نتیجه این مرحله در جدول ذیل ارائه شده است.

جدول ۳-۳۸ جدول تجزیه SLR(1)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	error	r4	r4		
2	error	r3	r3		
3					
4					
5					

حالت S3: حالت S3 شامل عنصر کاهشی .S→E است، در نتیجه، نماد شروع E تولید شده است، اگر نماد ورودی $ باشد، رشته ورودی نیز تمام شده است (زیرا$ نشان دهنده پایان رشته ورودی است)، در نتیجه رشته ورودی پذیرفته می‌گردد. در نتیجه:

action[3,$]= accept

با توجه به ماشین خودکار، عنصر E→E.+T در حالت S3 انتقالی است. در نتیجه نماد+ باعث تغییر حالت از حالت S3 به S4 می‌گردد و عنصر انتقالی، E→E.+T به عنصر E→E+.T تبدیل می‌گردد. این عمل همراه با انتقال + به پشته است. در نتیجه :

action[3,+]=S4

نماد id در این حالت بی تاثیر است در نتیجه اگر در حالت S3 نماد ورودی id باشد، خطا رخ می‌دهد، در نتیجه:

action[3,id]=error

جدول ۳-۳۹ جدول تجزیه SLR(1)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	error	r4	r4		
2	error	r3	r3		
3	error	s4	accept		
4					
5					

حالت S4: با توجه به ماشین خودکار، همه عناصر در حالت S4 انتقالی هستند، نمادهای T و id که بلافاصله بعد از علامت نقطه قرار دارند باعث تغییر حالت از حالت S4 می‌شوند. در ذیل به بررسی تاثیر هرکدام از این نمادها می‌پردازیم. با توجه به ماشین خودکار، نماد id باعث تغییر حالت از S4 به S1 می‌گردد، و عنصر انتقالی، T→.id به عنصر T→id. تبدیل می‌گردد. این عمل همراه با انتقال id به پشته است. در نتیجه action[4,id]=shift 1 است که به منظور اختصار action[4,id]=S1 است.

نمادهای + و $ در حالت S4 بی تاثیر هستند. اگر در حالت S4 نماد ورودی + یا $ باشد، خطا رخ داده است. در نتیجه:

action [4,+]=error
action[4,$]=error.

با توجه به ماشین خودکار، نمادT باعث تغییر حالت از S4 به S5 می‌گردد و عنصر انتقالی، E→E+.T به عنصر E→E+T. تبدیل می‌گردد. با توجه به اینکهT غیر پایانه است، در ورودی بی تاثیر است و فقط باعث تغییر حالت می‌گردد. در نتیجه action[4,T]=5 است.

جدول ۳-۴۰ جدول تجزیه SLR(1)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	error	r4	r4		
2	error	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5					

حالت S5: این حالت شامل عنصر کاهشی .E→E+T است. در این حالت یک دستگیره یافت شده است. اما این سوال مطرح می‌شود که در چه شرائطی باید عمل کاهش انجام شود؟ پاسخ آن است که اگر نماد جاری در ورودی نمادی باشد که می‌توانند بلافاصله بعد E قرار گیرد انجام کاهش بلامانع است. نمادهایی که می‌توانند بلافاصله بعد ازE قرار گیرند توسط follow(E) محاسبه می‌گردد. در نتیجه برای پایانه‌های follow(E) انجام کاهش بلامانع است.

follow(E)={+,$}

در نتیجه action[5,+]=reduce E→E+T و action[5,$]=reduce E→E+T است. به منظور اختصار در جدول تجزیه این موارد به صورت action[5,+]=r2 و action[5,$]=r2 نشان داده شده است، زیرا E→E+T قاعده تولید شماره 2 است. اگر نماد جاری id باشد خطا رخ می‌دهد در نتیجه action[2,id]=error .

مکانهای خالی را با error پر می‌کنیم.

جدول ۳-۴۱ جدول تجزیه SLR(1)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	error	r4	r4		
2	error	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5	error	r2	r2		

مثال ۳-٦١ برای درک بهتر نحوه استفاده از جدول تجزیه رشته id+id را توسط جدول تجزیه ۳-٤١ تجزیه می‌کنیم.

تجزیه رشته‌ها توسط جدول تجزیه SLR(1) مانند روش LR(0) است. به همین جهت از ارائه توضیحات خودداری کرده و فقط مراحل را نشان می دهیم. مراحل تجزیه id+id در جدول ۳-٤٢ نشان داده شده است. در ستون عملیات جدول ۳-۴۲ عمل استخراج شده از جدول تجزیه نشان داده شده است.

جدول ۳-۴۲ مراحل تجزیه رشته id+id

پشته	رشته ورودی	عملیات
0	id+id$	s1
0id1	+id$	r4: T→ id
0T	+id$	goto[0,T]=2
0T2	+id$	r3: E→ T
0E	+id$	goto[0,E]=3
0E3	+id$	s4
0E3+4	id$	s1
0E3+4id1	$	r4: T→ id
0E3+4T	$	goto[4,T]=6
0E3+4T6	$	r2: E→ E+T
0E	$	goto[0,E]=3
0E3	$	accept

روش ساخت جدول تجزیه SLR(1) با استفاده از ماشین خودکار را می‌توان به صورت ذیل خلاصه کرد.

- اگر در ماشین خودکار پایانه a باعث تغییر حالت از حالت Si به Sj گردد، action[i,a]=Sj را در قسمت action قرار می‌دهیم.

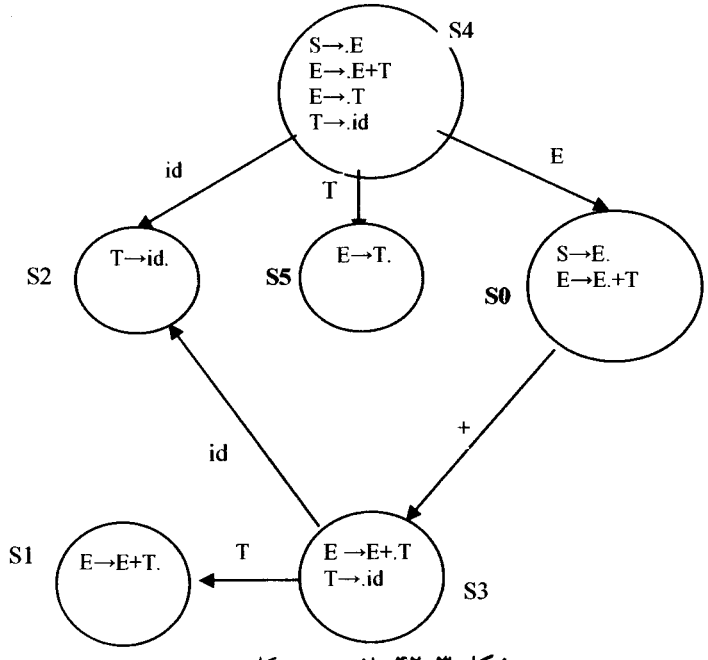
- اگر در ماشین خودکار غیر پایانه X باعث تغییر حالت از Si به حالت Sj گردد، goto[i,X]=j را در قسمت goto قرار می‌دهیم.

- اگر در ماشین خودکار، حالت Si شامل عنصر کاهشی.A→α باشد، (به جز عنصر کاهشی حاصل از قاعده تولید اضافه شده)، برای هر پایانه a در follow(A)، action[i,a]=rn می‌شود(rn به معنی کاهش با A→α است، r اختصار کلمه reduce و n شماره قاعده تولید A→α در گرامر است.)، زیرا عنصر کاهشی نشان دهنده یافتن دستگیره است و در نتیجه می‌توان کاهش را انجام داد.

- اگر در ماشین خودکار حالت Si شامل عنصر کاهشی قاعده تولید اضافه شده باشد (مانند.S→E ، به طوریکه E نماد شروع گرامر باشد) و نماد جاری $ باشد، action[i,$]=accept قرار می‌دهیم. اگر تجزیه کننده به این حالت برسد رشته ورودی پذیرفته می‌شود. زیرا کاهش S→E نماد شروع گرامر را تولید می‌کند و نماد $ نشان دهنده پایان رشته ورودی است، در نتیجه تجزیه با موفقیت تمام شده است.

- حالت که شامل عنصر قاعده تولید اضافه شده است حالت شروع ماشین خودکار است.

ترتیب شماره حالات ماشین خودکار در عملکرد تجزیه کننده بی تاثیر است به عنوان مثال ماشین خودکار گرامر ذیل را که در بخشهای قبلی ایجاد کردیم می توان به صورت زیر قرار داد.



شکل ۳-۴۲ ماشین خودکار

همانطور که ملاحظه می گردد شماره حالات تغییر کرده است. برای این ماشین خودکار نیز با استفاده از الگوریتم ارائه شده جدول تجزیه را به صورت ذیل تولید می کنیم.

جدول ۳-۴۳ جدول تجزیه

حالات	action			goto	
	id	+	$	T	E
0	error	s3	accept		
1	error	r2	r2		
2	error	r4	r4		
3	s2	error	r2	1	
4	s2	error	error	5	0
5	error	r3	r2		

حالت شروع حالتی است که شامل عنصر قاعده تولید اضافه شده باشد. به عنوان مثال در ماشین خودکار جدید حالت s4 حالت شروع است به همین جهت در فرایند تجزیه حالت شروع 4 است. به عنوان مثال رشته id+id را با استفاده از جدول تجزیه جدید تجزیه می‌کنیم.

جدول ۳-۴۴ مراحل تجزیه رشته id+id

پشته	رشته ورودی	عملیات
4	id+id$	S2
4id2	+id$	r4: T→ id
4T	+id$	goto[4,T]=5
4T5	+id$	r3: E→ T
4E	+id$	goto[4,E]=0
4E0	+id$	S3
4E0+3	id$	S2
4E0+3id2	$	r4: T→ id
4E0+3T	$	goto[3,T]=1
4E0+3T1	$	r2: E→ E+T
4E	$	goto[4,E]=0
4E0	$	Accept

همانطور که ملاحظه می گردد روند تجزیه تغییر نمی کند و فقط حالت شروع تغییر کرده است.

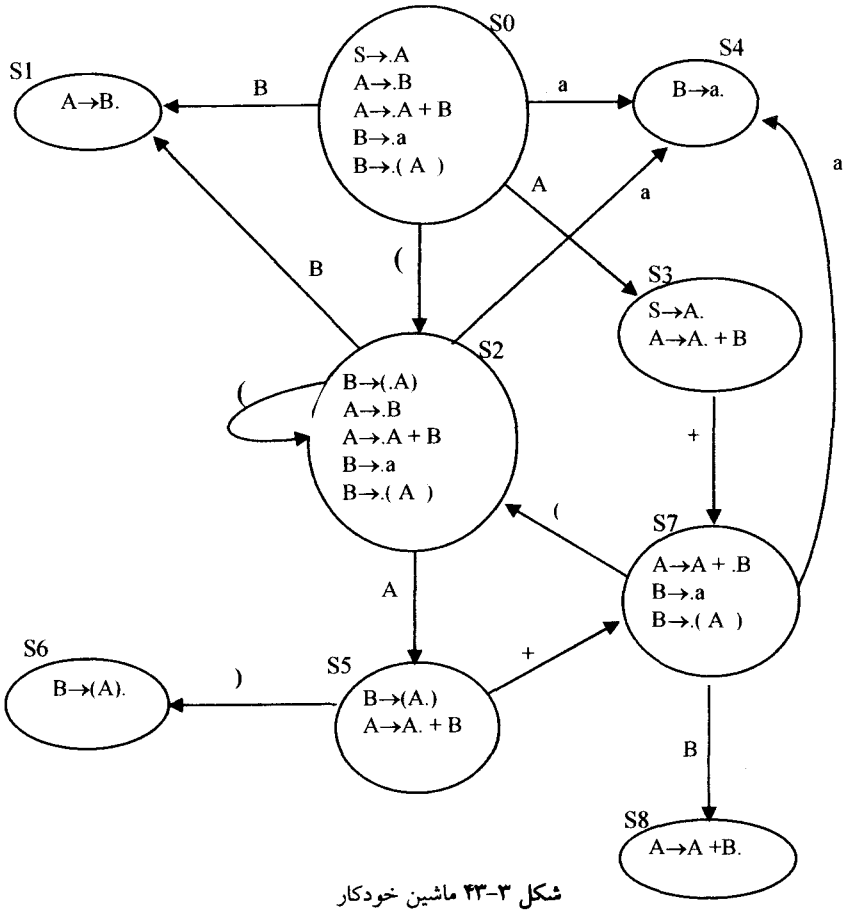
مثال۳-۶۲ جدول تجزیه SLR(1) گرامر ذیل را ایجاد کنید و رشته (a+a) را تجزیه کنید.

A→ B
A→ A + B
B→ a
B→ (A)

ابتدا قاعده تولید S→A را به گرامر اضافه کرده و قواعد تولید گرامر را شماره‌گذاری می‌کنیم. گرامر به صورت ذیل می‌گردد.

1- S→ A
2- A→ B
3- A→ A + B
4- B→ a
5- B→ (A)

برای گرامر جدید ماشین خودکار را رسم می‌کنیم. ماشین خودکار گرامر در شکل ۳-۴۳ نشان داده شده است. با توجه به ماشین خودکار شکل ۳-۴۳ جدول تجزیه ۳-۴۵ به دست می آید.



شکل ۳-۴۳ ماشین خودکار

جدول ۳-۴۵ جدول تجزیه

حالت	action					goto	
	a	+	(	)	$	A	B
0	s4		s2			3	1
1		r2		r2	r2		
2	s4		s2			5	1
3		s7			accept		
4		r4		r4	r4		
5		s7		s6			
6		r5		r5	r5		
7	s4		s2				8
8		r3		r3	r3		

با توجه به جدول تجزیه رشته (a+a) را تجزیه می‌کنیم.

جدول ۳-٤٦ مراحل تجزیه رشته (a+a)

پشته	رشته ورودی	اعمال انجام شده
0	(a+a)$	s2
0(2	a+a)$	s4
0(2a4	+a)$	r3:B→ a
0(2B1	+a)$	r2: A→ B
0(2A5	+a)$	s7
0(2A5+7	a)$	s4
0(2A5+7a4	)$	r4: B→a
0(2A5+7B8	)$	r3: A→ A + B
0(2A5	)$	s6
0(2A5)6	$	r5: B→ (A)
0A3	$	accept

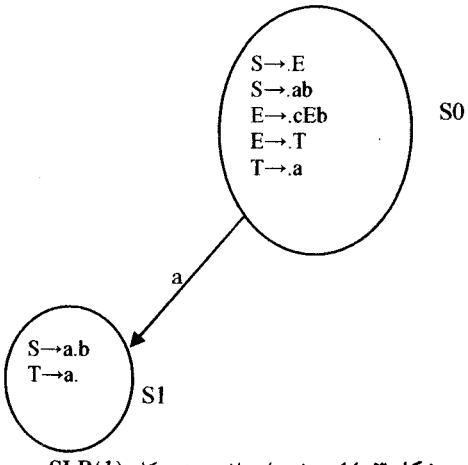
گرامرهای SLR(1)

گرامرهایی که برای آنها می‌توان جدول SLR(1) ایجاد کرد، گرامر SLR(1) می‌نامیم. اگر برای گرامری نتوان جدول تجزیه SLR(1) ایجاد کرد، گرامر SLR(1) نیست.

مثال۳-٦۳ گرامر ذیل را در نظر می‌گیریم.

S→ E | ab
E→cEb | T
T→a

بخشی از ماشین خودکار این گرامر به صورت ذیل است.



شکل ۳-٤٤ بخشی از ماشین خودکار SLR(1)

ماشین خودکار نشان می‌دهد که در حالت S1، عنصر .T→a کاهشی و S→a.b انتقالی است. b در Follow(T) قرار دارد در نتیجه کاهش .T→a مجاز است و همچنین b باعث انجام انتقال در عنصر S→a.b و تبدیل S→a.b به S→ab. می‌گردد. در نتیجه در حالت S1 مشخص نیست با نماد b آیا باید کاهش انجام گردد یا انتقال. این وضعیت برخورد انتقال/کاهش نامیده می‌شود. در نتیجه گرامر مورد نظر SLR(1) نیست.

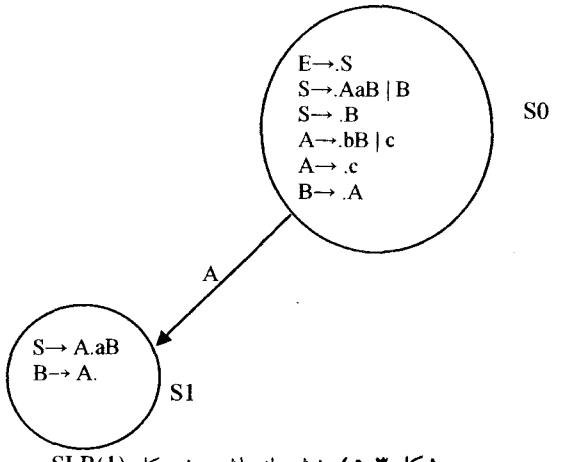
مثال۳-٦٤ نشان دهید گرامر ذیل SLR(1) نیست؟

S→AaB | B
A→bB | c
B→A

قاعده تولید E→S ، را به گرامر اضافه کرده و قواعد تولید را شماره گذاری می‌کنیم.

1- E→S
2- S→AaB
3- S→ B
4- A→bB
5- A→ c
6- B→A

بخشی از ماشین خودکار گرامر در شکل ۳-٤٥ ارائه شده است.



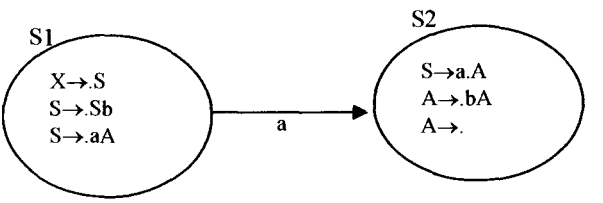
شکل ۳-٤٥ بخشی از ماشین خودکار SLR(1)

حالت S1 شامل، عنصر کاهشی .B→ A و عنصر انتقالی S→ A.aB است. کاهش.B→A برای پایانه‌های follow(B)={a,$} مجاز است. اما a باعث انتقال S→ A.aB نیز می‌شود و عنصر S→A.aB به S→Aa.B تبدیل می‌گردد. در نتیجه مشخص نیست در s1 با نمادa کدامیک از اعمال، انتقال S→A.aB و یا کاهش .B→A باید انجام شود. در نتیجه برخورد انتقال/کاهش رخ می‌دهد و گرامر SLR(1) نیست.

مثال۳-٦٥ آیا گرامر ذیل SLR(1) است.

S→Sb| aA
A→bA| ∈

ابتدا به گرامر قاعده تولید X→S را اضافه می‌کنیم و پس از شماره گذاری ماشین خودکار را رسم می‌کنیم. بخشی از ماشین خودکار در ذیل رسم شده است.



شکل ۳-٤٦ بخشی از ماشین خودکار SLR(1)

با توجه به حالت S2 عنصر A→. کاهشی است در نتیجه برای مجموعه follow(A) کاهش انجام می‌شود با توجه به گرامر b در مجموعه follow(A) است و همچنین باتوجه به A→.bA اگر نماد ورودی b باشد انتقال انجام می‌شود. این وضعیت نشان دهنده برخورد انتقال/کاهش است. در نتیجه گرامر SLR(1) نیست.

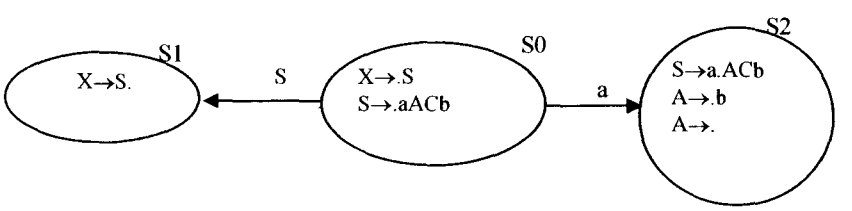
مثال ۳-٦٦ آیا گرامر ذیل SLR(1) است.

S→aACb
A→b | ∈
C→cC | ∈

ابتدا به گرامر قاعده تولید X→S را اضافه می‌کنیم.

X→S
S→aACb
A→b | ∈
C→cC | ∈

بخشی از ماشین خودکار گرامر به صورت ذیل است.



شکل ۳-٤٧ بخشی از ماشین خودکار SLR(1)

حالت S2 شامل عنصر کاهشی A→. است. کاهش این عنصر برای پایانه‌های مجموعه follow(A) مجاز است. با توجه به اینکه b در follow(A) است در نتیجه با نماد ورودی b

کاهش مجاز است . همچنین نماد b باعث باعث انتقال عنصر انتقالی A→.b می‌شود. بنابراین با نماد ورودی b کاهش عنصر .A→ و انتقال A→.b هر دو مجاز است در بنابراین برخورد انتقال/ کاهش رخ می‌دهد بنابراین گرامر SLR(1) نیست.

مقایسه جدول LR(0) و SLR(1)

در جدول LR(0) هرگاه یک عنصر کاهشی به صورت A→α مشاهده شود کاهش انجام می‌گردد، در حالیکه در جدول SLR(1) برای پایانه‌های follow(A) اجازه کاهش داده شده است و در نتیجه برای برخی از پایانه‌ها اجازه کاهش داده نشده است.

مثال۳-۶۷ جدول تجزیه LR(0) و SLR(1) برای گرامر ذیل داده شده است.

1- S→ E
2- E→E+T
3- E→T
4- T→id

جدول ۳-۴۷ جدول تجزیه LR(0)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	r4	r4	r4		
2	r3	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5	r2	r2	r2		

جدول ۳-۴۸ جدول تجزیه SLR(1)

حالات	action			goto	
	id	+	$	T	E
0	s1	error	error	2	3
1	error	r4	r4		
2	error	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5	error	r2	r2		

همانطور که ملاحظه می‌گردد تعداد سطرهای هر دو جدول یکسان است. اما در LR(0) برای هر حالت که شامل عنصر کاهشی است (مانند حالات S1,S2,S5) صرفنظر از پایانه ورودی کاهش انجام شده است، در حالیکه در جدول SLR(1) برای پایانه‌های مجموعه follow کاهش انجام شده است.

مثال ۳-۶۸ برای مقایسه بهتر دو روش رشته idid را با هر دو روش تجزیه می‌کنیم.

جدول۳-۴۹ تجزیه رشته idid با استفاده از جدول LR(0)

پشته	رشته ورودی	اعمال انجام شده
0	idid$	s1
0id1	id$	r4
0T2	id$	r3
0E3	id$	error

جدول ۳-۵۰ تجزیه رشته idid با استفاده از جدول تجزیه SLR(1)

پشته	رشته ورودی	اعمال انجام شده
0	idid$	s1
0id1	id$	error

همانطور که دو جدول بالا نشان می‌دهد SLR(1) سریعتر از LR(0) خطا را کشف می‌کند. روش SLR(1) از LR(0) قوی تر است در نتیجه اگر گرامری SLR(1) نباشد، LR(0) نیز نیست و همچنین اگر گرامری LR(0) باشد، SLR(1) نیز هست. روش SLR(1) نسبت به LR(0) تعداد بیشتری از گرامرها را پوشش می‌دهد.

۳-۱۸-۳- روش LR(1)

در SLR(1) کاهش یک عنصر کاهشی مانند $N\rightarrow\alpha.$ ، در صورتی مجاز است که نماد ورودی در مجموعه follow(N) باشد. اما این روش دقیق نیست، زیرا یک غیرپایانه ممکن است شامل چند قاعده تولید به صورت ذیل باشد.

$N\rightarrow\alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$

follow(N) شامل پایانه‌هایی است که با استفاده از $\alpha_1, \alpha_2, ..., \alpha_n$ به دست آمده اند. در حالیکه یک عنصر کاهشی فقط با استفاده از یکی از $\alpha_1, \alpha_2, ..., \alpha_n$ به دست می‌آید. به عبارت دیگر ممکن است یک عنصر کاهشی که با استفاده از α_1 به دست آمده است، با پایانه‌ای از

follow(N) که با استفاده از α_2 به دست آمده است کاهش یابد، که کاهش درستی نیست. برای حل این مشکل باید هر عنصری پایانه‌های مورد انتظار را با خود به همراه داشته باشد. یعنی در زمانی که عنصر مربوط به α_1 تولید می‌شود، پایانه‌های مورد انتظار به همراه آن باشد و در مورد α_2 تا α_n نیز همین روال تکرار می‌گردد. در این صورت از ترکیب پایانه‌ها جلوگیری می‌گردد. پایانه‌های همراه عناصر را نماد پیشگو می‌نامیم. روش LR(1) نماد بعدی را پیشگویی می‌کند، اگر هنگام کاهش نماد جاری ورودی با نماد پیشگوئی شده برابر باشد کاهش انجام می‌شود، در غیر این صورت کاهش انجام نمی‌شود.

اگر S نماد شروع گرامر و A قاعده تولید اضافه شده و w رشته ورودی باشد، با توجه به اینکه انتهای رشته w علامت $ قرار دارد، رشته w به w$ تبدیل می‌گردد. اولین پیشگویی به این صورت است که بعد از کاهش w به S نماد ورودی $ است. زیرا با کاهش w به S رشته w$ به S$ تبدیل می‌گردد. این پیشگویی را با عنصری به صورت ذیل نشان می‌دهیم.

[A→.S , $]

عنصر [A→.S,$]، وجود یک $ را برای کاهش S به A پیشگویی می‌کند. به عبارت دیگر بعد از تولید S و تبدیل [A→.S,$] به [A→S.,$]، کاهش عنصر کاهشی .A→S وقتی مجاز است که نماد جاری در ورودی $ باشد. همانطور که ملاحظه می گردد در LR(1) نمادهای قابل قبول همراه عنصر است.

با استفاده از یک عنصر پیشگویی، پیشگوییهای دیگری را می‌توان نتیجه گرفت. این عناصر را عناصر LR(1) می‌نامیم، زیرا این عناصر یک نماد را پیشگویی می‌کنند.

در روش LR(0) که در بخشهای قبلی تشریح شد هیچ نمادی پیش‌گویی نمی‌شد به همین جهت آن را LR(0) نامیدیم. در روش SLR(1) یک نماد ورودی برای مجاز بودن کاهش استفاده می‌گردد به همین جهت عدد یک در SLR(1) درج می‌شود. در روش LR(1) از یک نماد پیشنگر استفاده می‌گردد به همین جهت آن را LR(1) می‌نامیم. روشهایی وجود دارند که بیش از یک نماد ورودی استفاده می‌کنند این گونه روشها را LR(K) می‌نامیم. K نشان دهنده تعداد نمادهای ورودی است که به عنوان پیشنگر استفاده می‌شوند.

همانند روش LR(0) و SLR(1)، ابتدا عناصر را تولید می‌کنیم و با گروهبندی آنها یک ماشین خودکار ایجاد می‌کنیم که برای تولید جدول تجزیه استفاده می‌گردد. به منظور درک بهتر LR(1)، ماشین خودکاری برای گرامر ذیل ایجاد می‌کنیم.

S→ E | ab
E→ dEb | T
T→ a

مانند LR(0) و SLR(1) ابتدا قاعده تولید جدید به گرامر اضافه کرده و قواعد تولید را شماره گذاری می‌کنیم. گرامر جدید به صورت ذیل است.

1- A→S
2- S→ E
3- S→ ab
4- E→ dEb
5- E→ T
6- T→ a

با توجه به نماد شروع گرامر جدید، اولین پیشگویی به این صورت است: بعد از کاهش S به A نماد ورودی $ است زیرا با کاهشw به S رشته پایان می‌یابد که در این صورت در ورودی باید $ دیده شود. این پیشگویی را به صورت ذیل نشان می‌دهیم.

[A→.S , $]

وقتی نقطه به انتهای عنصر رسید و عنصر [A→S.,$] به دست آمد، در صورتی کاهش S به A مجاز است که نماد ورودی $ باشد. بر اساس اولین پیشگویی می‌توان پیشگویی‌های دیگری را انجام داد و عناصر جدید را تولید کرد.

عنصر [A→.S,$] بیان می‌کند که کاهش S به A وقتی مجاز است که نماد ورودی $ باشد به عبارت دیگر بعد از S نماد $ قرار داشته باشد. در این عنصر علامت نقطه قبل از S قرار دارد بنابراین ابتدا باید S تولید گردد. برای تولید S دو راه با استفاده کاهش ab به S (با توجه به قاعده تولید S→ab) و کاهش E به S (با توجه به قاعده تولید S→E) وجود دارد. در نتیجه پس از کاهش ab به S نماد ورودی نیز $ است. بنابراین کاهش ab به S وقتی مجاز است که نماد ورودی $ باشد. با توجه به اینکه هنوز ab کاهش نیافته است علامت نقطه در ابتدای ab قرار دارد. این پیشگویی را با [S→.ab,$] نشان می‌دهیم. همچنین پس از کاهش E به S (با استفاده از S→E) نیز نماد ورودی باید $ باشد. با توجه به اینکه E هنوز کاهش نیافته است علامت نقطه قبل از E قرار دارد. این پیشگویی را با [S→.E,$] نشان می‌دهیم. در نتیجه پیشگویی‌های ذیل به دست می‌آید.

[A→.S , $]
[S→.ab,$]
[S→.E,$]

با استفاده از پیشگوئی [S→.E,$] می‌توان پیشگویی‌های دیگری استنتاج کرد. عنصر [S→.E,$] بیان می‌کند که کاهش E به S وقتی مجاز است که نماد ورودی $ دیده شود. علامت نقطه قبل از E قرار دارد بنابراین باید E تولید گردد تا [S→.E,$] به [S→E.,$] تبدیل می گردد. دو راه برای تولید E با استفاده از کاهش T به E (با توجه به قاعده تولید E→T) و کاهش dEb به E (با توجه به قاعده تولید E→dEb) وجود دارد. بنابراین بعد از کاهش T به E نماد ورودی باید $ باشد. چون هنوز T کاهش تولید نشده است بنابراین علامت نقطه قبل از T

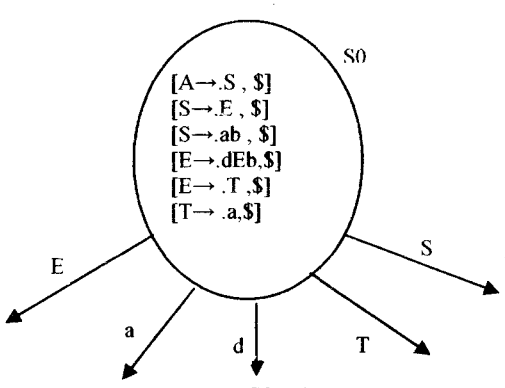
قرار دارد. این پیشگویی را با عنصر [E→.T,$] نشان می‌دهیم. همچنین بعد از کاهش dEb به E نیز نماد ورودی باید $ باشد این پیشگویی را [E→.dEb,$] نشان می‌دهیم. در نتیجه مجموعه عناصر به شرح ذیل است.

[A→.S , $]
[S→.E , $]
[S→.ab , $]
[E→ .T ,$]
[E→ .dEb ,$]

با استفاده از پیشگوییهای به دست آمده سعی می‌کنیم، پیشگوییهای دیگری را انجام دهیم. عنصر [E→.T,$] پیشگویی می‌کند که بعد از کاهش T باید $ در ورودی باشد با توجه به قاعده تولید T→a می‌توان عنصر [T→.a,$] را نتیجه گرفت. در نتیجه مجموعه پیشگوییها به صورت ذیل خواهد شد.

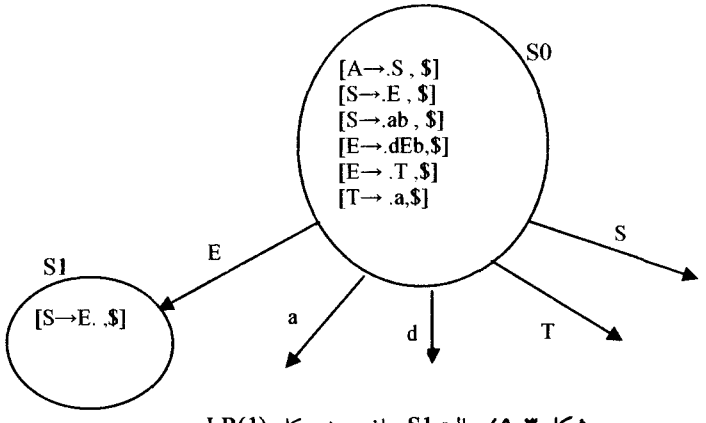
[A→.S , $]
[S→.E , $]
[S→.ab , $]
[E→ .T ,$]
[E→ .dEb ,$]
[T→ .a ,$]

با بررسی بیشتر این مجموعه، پیشگویی جدیدی به دست نمی‌آید. این مجموعه را S0 می‌نامیم. S0 یکی از حالات ماشین خودکار است. علائم E,a,d,T و S که سمت راست نقطه قرار دارند نشان دهنده انواع انتقالهای ممکن از S0 است. نتیجه این بررسی در شکل ذیل نشان داده شده است.



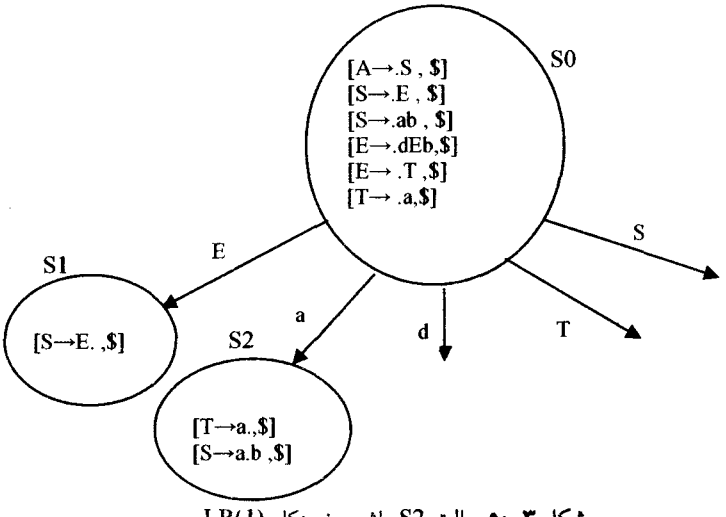
شکل ۳-۴۸ حالت S0 ماشین خودکار LR(1)

نماد E باعث تبدیل عنصر [S→.E,$] به [S→E.,$] می‌گردد. این عنصر کاهشی است در نتیجه پیشگویی جدیدی قابل استخراج نیست. مجموعه {[S→E.,$]} را S1 می‌نامیم. نتیجه در شکل ذیل ارائه شده است.



شکل ۳-۴۹ حالت S1 ماشین خودکار LR(1)

نماد a باعث تبدیل عنصر [$,S→.ab] به [$,S→a.b] و تبدیل عنصر [$,T→.a] به [$,T→a.] می‌گردد. عنصر [$,T→a.] کاهشی است در نتیجه عنصر جدیدی از آن قابل استنتاج نیست. در عنصر [$,S→a.b] نماد بعد از نقطه یک پایانه است در نتیجه عنصر جدیدی قابل استنتاج نیست. مجموعه عناصر {[$,T→a.] , [$,S→a.b]} را S2 می‌نامیم. نتیجه در شکل ذیل ارائه شده است.



شکل ۳-۵۰ حالت S2 ماشین خودکار LR(1)

مجموعه S2 شامل یک عنصر کاهشی و یک عنصر انتقالی است. در روش SLR(1) این حالت باعث برخورد انتقال/کاهش می‌گردید. زیرا b باعث انتقال در عنصر T→a.b می‌شود و چون

b در Follow(T) است باعث کاهش عنصر .T→a می‌گردد. ولی بر خلاف روش SLR(1) روش LR(1) عناصر S2 موجب برخورد انتقال/کاهش نمی‌شوند. زیرا با توجه به [T→a.,$] کاهش زمانی انجام می‌شود که نماد جاری $ باشد و با توجه به [S→a.b,$] انتقال زمانی انجام می‌شود که نماد جاری b باشد. در نتیجه برخوردی رخ نمی‌دهد. زیرا اگر نماد جاری $ باشد کاهش و اگر b باشد، انتقال انجام می‌شود.

نماد d باعث می‌گردد عنصر [E→.dEb,$] به [E→d.Eb,$] تبدیل گردد. با استفاده از عنصر پیشگویی [E→d.Eb,$] سعی می‌کنیم پیشگوییهای دیگری بیابیم. در عنصر [E→d.Eb,$] نقطه قبل از E قرار دارد. بنابراین پیشگویی بعدی از E به دست می‌آید.

در عنصر [E→d.Eb,$] بلافاصله بعد از E نماد b قرار دارد. با توجه به قاعده تولید E→T می‌توان T را به E کاهش داد. در نتیجه بعد از کاهش T به E نماد ورودی باید b باشد. بنابراین کاهش E→T وقتی صحیح است که پایانه ورودی b باشد. این پیشگویی را با عنصر[E→.T,b] نشان می‌دهیم. مجموعه پیشگویی‌های جدید به دست آمده عبارتند از:

[E→ d.Eb ,$]
[E→ .T , b]

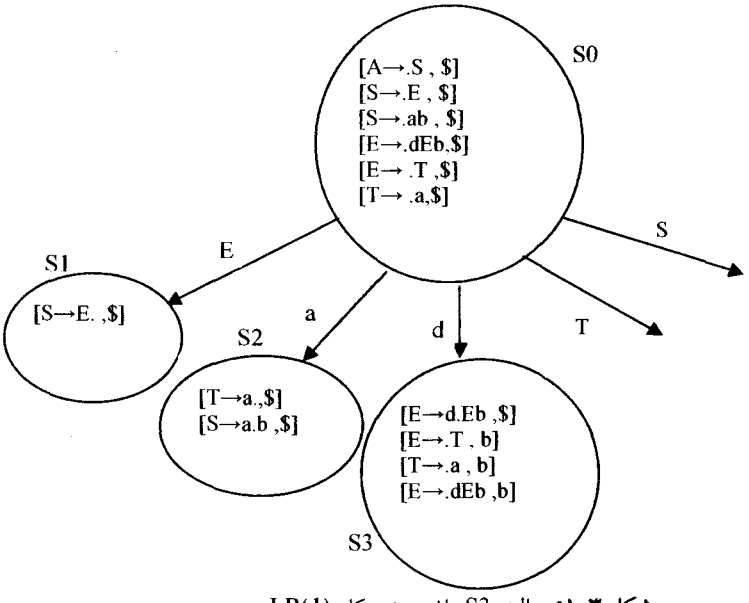
در عنصر [E→d.Eb,$] بلافاصله بعد از E نماد b قرار دارد. برای تولید E با توجه به قاعده تولید E→dEb می‌توان dEb را به E کاهش داد. بعد از کاهش dEb به E نماد ورودی باید b باشد. بنابراین کاهش dEb به E وقتی صحیح است که پایانه ورودی b باشد. این پیشگویی را با عنصر [E→.dEb,b] نشان می‌دهیم. مجموعه پیشگویی‌های جدید به دست آمده عبارتند از:

[E→d.Eb ,$]
[E→.T , b]
[E→.dEb ,b]

عنصر [E→.T,b] بیان می‌کند که پس از کاهش T نماد ورودی باید b باشد. برای تولید T یک راه با توجه به T→a وجود دارد. بنابراین پس از کاهش a به T نماد ورودی باید b باشد. در نتیجه عنصر [T→.a , b] به دست می‌آید. مجموعه پیشگویی‌ها عبارتند از:

[E→d.Eb ,$]
[E→.T , b]
[E→.dEb ,b]
[T→.a , b]

با بررسی بیشتر عناصر این مجموعه، پیشگویی دیگر قابل استنتاج نیست. مجموعه عناصر {[E→d.Eb,$], [E→.dEb,b],[E→.T,b],[T→.a,b]} را S3 می‌نامیم. نتیجه در شکل ذیل نشان داده شده است.



شکل ۳-۵۱ حالت S3 ماشین خودکار LR(1)

نمادهای پیشگویی ممکن است شامل چند نماد باشد. برای محاسبه مجموعه عناصر پیشگویی قابل استنتاج از عنصر I مانند روش LR(0) می‌توان از closure(I) استفاده کرد. به منظور محاسبه closure(I) برای روش LR(1) ، از قوانین ذیل تا زمانیکه عنصری به مجموعه closure(I) اضافه می‌کند استفاده می‌شود.

۱- I به closure(I) اضافه می‌گردد.

۲-اگر [p→α.Nβ,a] یک عنصر و N→γ نیز یک قاعده تولید و c در first(βa) باشد آنگاه عنصر [N→.γ,c] باید به مجموعه عناصر اضافه گردد.

اثبات: این مسئله را در دو قسمت اثبات می‌کنیم.

۱- با توجه به عنصر [p→α.Nβ,a] علامت نقطه قبل از N قرار دارد بنابراین در این وضعیت باید N تولید گردد. برای تولید N می توان γ را به N کاهش داد (با توجه به قاعده تولید N→γ). با توجه به اینکه بعد از N نماد β قرار دارد در نتیجه بعد از کاهش γ به N نمادهای ورودی نمادهایی هستند که β با آنها شروع می شوند یا به عبارتی first(β) هستند.

۲- اگر β بتواند تهی گردد در این صورت [p→α.Nβ,a] به [p→α.N,a] تبدیل می‌شود. [p→α.N,a] بیان می‌کند که برای کاهش αN به p نماد ورودی a است. با توجه به قاعده تولید

N→γ نماد N را می‌توان از کاهش γ به N تولید کرد که در این صورت بعد از γ نیز نماد ورودی باید a باشد. به عبارت دیگر کاهش γ به N وقتی مجاز است که نماد ورودی a باشد. با توجه به ۱ و۲ ، کاهش N→γ در صورتی صحیح است که نماد جاری در رشته ورودی در First(βa) باشد. این پیشگویی را با عنصری به صورت [N→.γ,c] نشان می‌دهیم به طوریکه c متعلق به First(βa) باشد.

مثال ۳-۶۹ closure([E→ d.Eb ,$]) را محاسبه کنید.

- [E→ d.Eb ,$] به مجموعه closure([E→d.Eb ,$]) اضافه می‌شود، بنابراین:

closure([E→ d.Eb ,$])={ [E→ d.Eb ,$]}

- عنصر [E→ d.Eb ,$] در closure([E→ d.Eb ,$]) و E→T در گرامر و first(b$)={b} است ، در نتیجه [E→.T,b] به مجموعه closure([E→d.Eb,$]) اضافه می‌شود بنابراین:

closure([E→ d.Eb ,$])={ [E→ d.Eb ,$], [E→.T ,b] }

- عنصر [E→.T,b] در closure([E→d.Eb ,$]) و T→a در گرامر و first(b)={b} است، (در این قسمتβ تهی است)، در نتیجه[T→.a,b] به مجموعه closure([E→d.Eb,$]) اضافه می‌شود بنابراین:

closure([E→d.Eb ,$])={[E→d.Eb,$], [E→.T,b], [T→.a,b]}

- عنصر [E→d.Eb,$] در closure([E→d.Eb,$]) وE→dEb در گرامر و first(b$)={b} است، در نتیجه [E→.dEb,b] به مجموعه closure([E→d.Eb,$]) اضافه می‌شود بنابراین:

closure([E→d.Eb ,$])={[E→d.Eb,$], [E→.T,b], [T→.a,b], [E→.dEb,b] }

با بررسی بیشتر این مجموعه، عنصر جدیدی اضافه نخواهد شد. این مجموعه عناصر حالت S3 در ماشین خودکار است. با استفاده از closure ماشین خودکار LR(1) گرامر به صورت ذیل است.

با استفاده از ماشین خودکار شکل ۳-۵۲ می‌توان جدول تجزیه را ایجاد کرد. تفاوت ساخت جدول تجزیه از ماشین خودکار LR(1) و SLR(1) در شرایط کاهش است. در SLR(1) کاهش S→α. زمانی انجام می‌شود که نماد جاری در follow(S) باشد اما در LR(1) کاهش [S→α.,l] زمانی انجام می‌شود که نماد جاری یکی از نمادهای پیشگویی شده، یعنی یکی از نمادهای مجموعه l باشد. به عنوان مثال حالت S1 شامل عنصر [S→E.,$] است.

عنصر [S→E., $] نشان می‌دهد کاهش S→E زمانی صحیح است که نماد جاری $ باشد. در نتیجه:

action[1,$]= reduce S→E

یا

action[1,$]= r2

حالت S2 شامل عنصر [S→a.b, $] و [T→ a.,$] است. [T→a.,$] نشان می‌دهد کاهش T→a زمانی صحیح است که نماد جاری $ باشد. در نتیجه:

action[2,$]= reduce T→a

یا

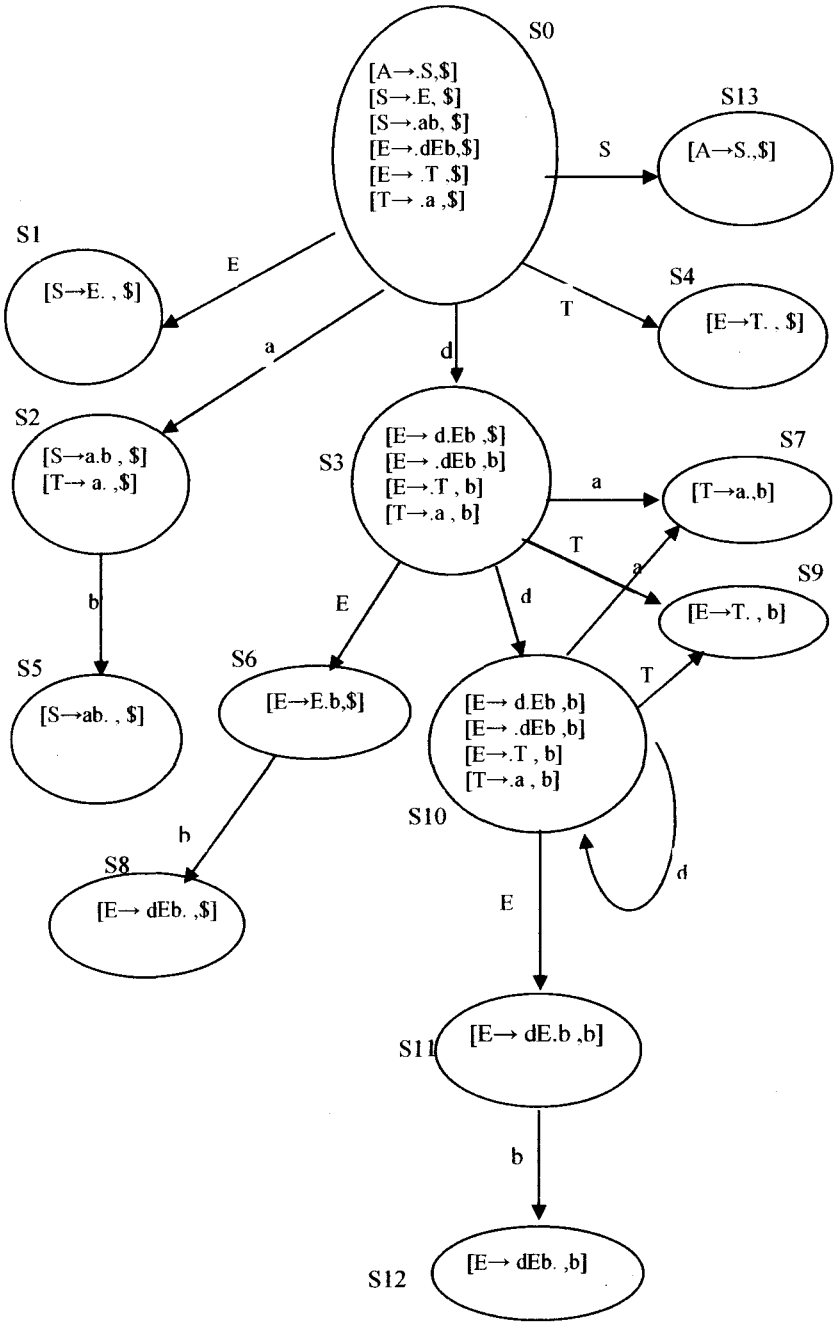
action[2,$]= r6

با توجه به ماشین خودکار و[S→a.b, $] ، نماد b باعث تغییر حالت از S2 به S5 می‌گردد، در نتیجه:

action[2,b]=Shift 5

یا

action[2,b]=S5



شکل ۳-۵۲ ماشین خودکار LR(1)

روش ساخت جدول تجزیه LR(1) را می‌توان به صورت ذیل خلاصه کرد.

- اگر در ماشین خودکار پایانه a باعث تغییر حالت از حالت Si به Sj گردد، action[i,a]=Sj را در قسمت action قرار می‌دهیم.

- اگر در ماشین خودکار غیر پایانه X باعث تغییر حالت از Si به Sj گردد، goto[i,X]=j را در قسمت goto قرار می‌دهیم.

- اگر در ماشین خودکار، حالت Si شامل عنصر کاهشی[A→α.,L] باشد، (به جز عنصر کاهشی حاصل از قاعده تولید اضافه شده)، برای هر پایانه در L action[i,a]=rn می‌شود(rn به معنی کاهش با A→α است، r اختصار کلمه reduce و n شماره قاعده تولید A→α در گرامر است).

- اگر در ماشین خودکار، حالت Si شامل عنصر کاهشی قاعده تولید اضافه شده باشد (مانند[S→E.,$]، به طوریکه E نماد شروع گرامر است و نماد جاری $ باشد، action[i,$]=accept قرار می‌دهیم. اگر تجزیه کننده به این حالت برسد رشته پذیرفته می‌شود.

- قسمتهای خالی را با error پر می‌کنیم و یا برای اختصار خالی قرار می‌دهیم.

- حالت که شامل عنصر قاعده تولید اضافه شده است حالت شروع ماشین خودکار است.

در ذیل برای نمونه چند خانه از جدول را پر می‌کنیم.

مثال۳-۷۰ حالت S7 شامل عنصر کاهشی [T→a.,b] است، کاهش T→a فقط برای نماد پیشگویی شده b مجاز است در نتیجه:

action[7,b]=r6

مثال۳-۷۱ نماد d باعث تغییر حالت از S3 به S10 می‌شود. بنابراین:

action[3,d]=s10

مثال۳-۷۲ نماد d باعث تغییر حالت از S10 به S10 می‌شود. بنابراین:

action[10,d]=s10

مثال۳-۷۳ غیر پایانه T باعث تغییر حالت از S3 به S9 می‌شود. چون T غیرپایانه است بنابراین:

goto[3,T]=9

مثال۳-۷٤ حالت S13 که شامل عنصر شروع [A→S.,$] است را در نظر می‌گیریم. با توجه به عنصر کاهشی [A→S.,$] انجام کاهش فقط با رویت نماد $ در ورودی مجاز است، که در این صورت رشته به نماد شروع گرامر اصلی یعنی S کاهش یافته است و رشته ورودی نیز تمام گشته است. در نتیجه action[13,$]=accept.

با توجه به ماشین خودکار جدول تجزیه به صورت ذیل است.

جدول ۳-۵۱ جدول تجزیه LR(1)

حالات	action				goto		
	a	b	d	$	E	T	S
0	s2		s3		1	4	13
1	r2			r2			
2		s5		r6			
3	s7		s10		6	9	
4				r5			
5				r3			
6		s8					
7		r6					
8				r4			
9		r5					
10	s7		s10		11	9	
11		s12					
12		r4					
13				accept			

مثال۳-۷۵ نشان دهید گرامر ذیل LR(1) است و جدول تجزیه آن را رسم کنید.

A→Bb|cBd|ed|cba
B→e

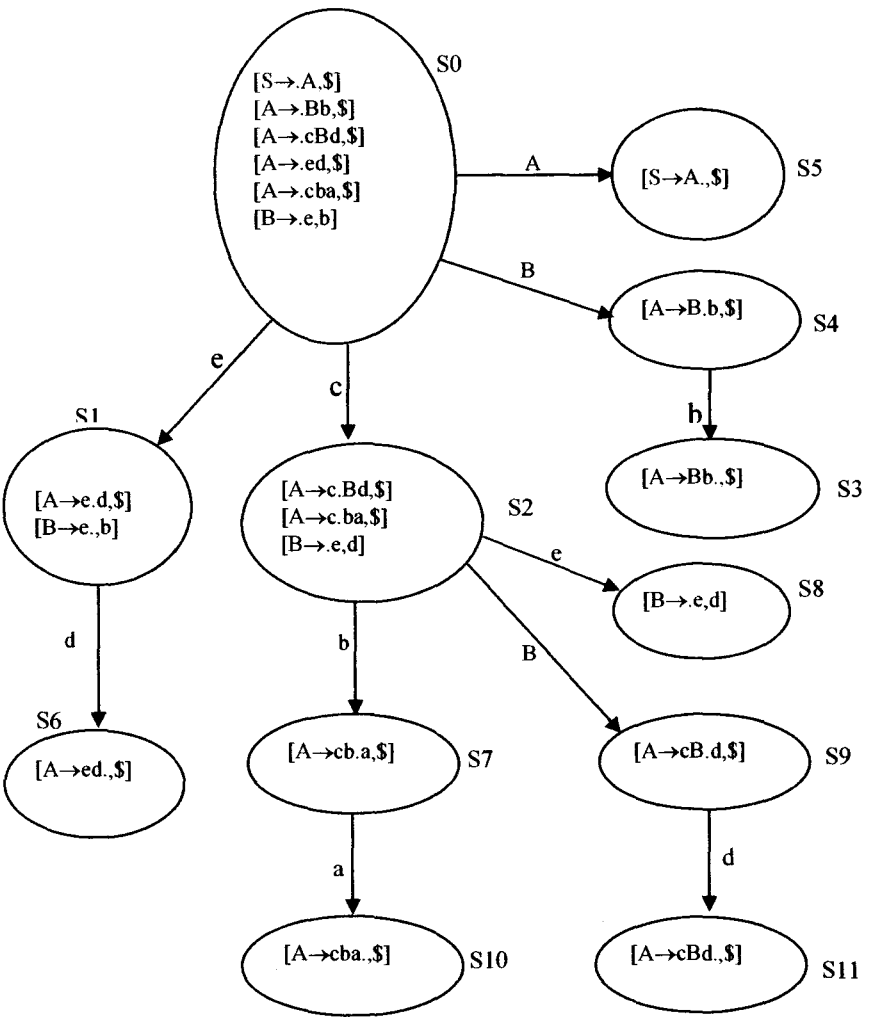
ابتدا قاعده تولید S→A را به گرامر اضافه می‌کنیم و قواعد را شماره گذاری می‌کنیم.

1- S→ A
2- A→ Bb
3- A→ cBd
4- A→ ed
5- A→ cba
6- B→e

ماشین خودکار در شکل ۳-۵۳ و جدول تجزیه در جدول ۳-۵۲ نشان داده شده است.

جدول ۳-۵۲ جدول تجزیه LR(1)

حالات	action						goto	
	a	b	c	d	e	$	A	B
0			s2		s1		5	4
1		r6		s6				
2		s7			s8			9
3								
4		s3						
5						accept		
6						r4		
7	s10							
8				r6		r6		
9				s11				
10						r5		
11						r3		



شکل ۳-۵۳ ماشین خودکار LR(1)

مثال۳-۷۶ نشان دهید گرامر ذیل LR(1) است و جدول تجزیه آن را رسم کنید.

A→BB
B→bB|a

ابتدا قاعده تولید S→A را به گرامر اضافه می‌کنیم و قواعد را شماره‌گذاری می‌کنیم.

1- S→ A
2- A→ BB
3- B→ bB
4- B→ a

اولین عنصر [S→.A,$] است. (closure([S→.A,$]) را در ذیل محاسبه می‌کنیم.

- [S→.A,$] به (closure([S→.A,$]) اضافه می‌شود. در نتیجه:

closure([S→.A,$])={ [S→.A,$]}

- [S→.A,$] در (closure([S→.A,$]) و A→BB در گرامر و first($)={$} است (با توجه به قوانین محاسبه Closure در اینجا β تهی است.). بنابراین [A→.BB,$] به (closure([S→.A,$]) اضافه می‌شود در نتیجه:

closure([S→.A,$])={ [S→.A,$], [A→.BB,$]}

- [S→.BB,$] در (closure([S→.A,$]) و B→bB در گرامر و first(B$)={a,b} است (با توجه به قوانین محاسبه Closure در اینجا β=B است) بنابراین دو عنصر [B→.bB,a] و[B→.bB, b] به دست می‌آید. برای اختصار نمادهای پیشگویی این دو عنصر را ترکیب می‌کنیم، در نتیجه عنصر [B→.bB,ab] به (closure([S→.A,$]) اضافه می‌شود، (عنصر [B→.bB,ab] دو عنصر پیشگویی a و b دارد.) در نتیجه:

closure([S→.A,$])={ [S→.A,$], [S→.BB,$], [B→.bB,ab] }

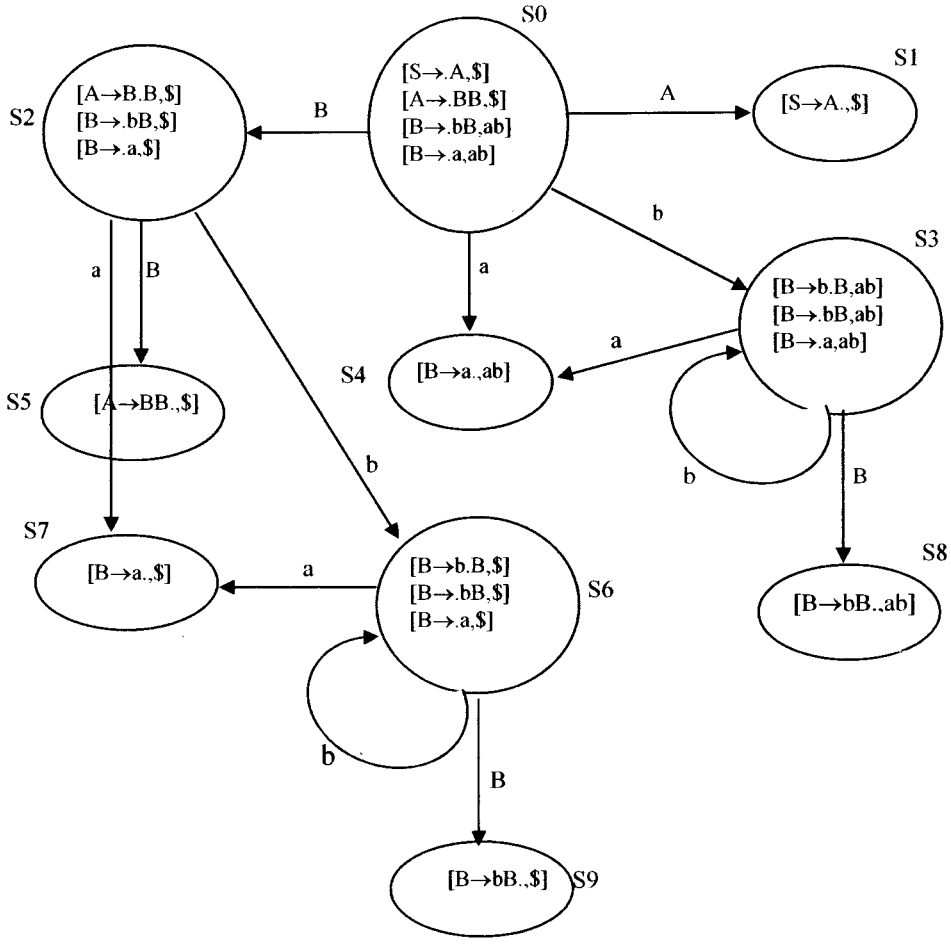
- [A→.BB,$] در (closure([S→.A,$]) و B→a در گرامر و first(B$)={a,b} است بنابراین دوعنصر [B→.a,b] و[B→.a,a] به دست می‌آید. برای اختصار نمادهای پیشگویی دو عنصر [B→.a,b] و[B→.a,a] را ترکیب می‌کنیم، در نتیجه عنصر [B→.a,ab] به دست می‌آید. [B→.a,ab] به (closure([S→.A,$]) اضافه می‌شود در نتیجه:

closure([S→.A,$])={ [S→.A,$], [A→.BB,$], [B→.bB,ab], [B→.a,ab] }

مجموعه (closure([S→.A,$]) را S0 می‌نامیم.

نماد a باعث می‌شود [B→.a,ab] به عنصر کاهشی [B→a.,ab] تبدیل می‌شود. چون سمت راست نقطه نمادی وجود ندارد در نتیجه پیشگویی جدیدی قابل استنتاج نیست. این مجموعه را S4 می‌نامیم.

مانند حالت S0 حالت دیگر ماشین خودکار را می سازیم. ماشین خودکار کامل در شکل ۳-۵۱ ارائه شده است. پس از رسم ماشین خودکار جدول تجزیه را رسم می کنیم. به عنوان مثال عنصر کاهشی حالت S4 شامل دو نماد پیشگویی a و b است. بنابراین action[4,a]=r3 و action[4,b]=r3 است. جدول تجزیه در جدول ۳-۵۳ ارائه شده است.



شکل ۳-۵۴ ماشین خودکار LR(1)

جدول ۳-۵۳ جدول تجزیه LR(1)

حالت	action			goto	
	b	a	$	A	B
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

برای تعیین LR(1) بودن گرامر مانند روش LR(0) و SLR(1) اگر برخورد انتقال/کاهش و کاهش/کاهش رخ دهد، گرامر LR(1) نیست.

مثال۳-۷۷ ماشین خودکار LR(1) گرامر ذیل را رسم کنید.

E→E+E|E-E|a

قاعده تولید S→E را به گرامر اضافه کرده و قواعد تولید را شماره گذاری می‌کنیم.

1- S→E
2- E→E+E
3- E→E-E
4- E→a

اولین عنصر [S→.E,$] است. closure([S→.E,$]) را به صورت ذیل محاسبه می‌کنیم.

- عنصر [S→.E,$] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$]}

- [S→.E,$] در closure([S→.E,$]) و E→E+E در گرامر و first($)={$} است (با توجه به قوانین محاسبه Closure در اینجا β تهی است.). بنابراین [E→.E+E,$] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$], [E→.E+E,$] }

- [S→.E,$] در closure([S→.E,$]) و E→E-E در گرامر و first($)={$} است بنابراین [E→.E-E,$] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$], [E→.E+E,$] , [E→.E-E,$] }

- [S→.E,$] در closure([S→.E,$]) و E→a در گرامر و first($)={$} است بنابراین [E→.a,$] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$], [E→.E+E,$] , [E→.E-E,$] , [E→.a,$] }

- [S→.E+E,$] در closure([S→.E,$]) و E→a در گرامر و first(+E$)={+} است (با توجه به قوانین محاسبه Closureدر اینجا β=+Eاست.). بنابراین [E→.a,+] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$], [E→.E+E,$] , [E→.E-E,$] , [E→.a,$], [E→.a,+] }

نمادهای پیشگویی دو عنصر [E→.a,$] ,[E→.a,+] را ترکیب می‌کنیم. بنابراین عنصر [E→.a,$+] را به جای دوعنصر [E→.a,$] ,[E→.a,+] به مجموعه اضافه می‌کنیم. درنتیجه:

closure([S→.E,$])={[S→.E,$], [E→.E+E,$] , [E→.E-E,$] , [E→.a,$+] }

دقت کنید که در [E→.a,$+] ترتیب $+ مهم نیست زیرا این نمادها مجموعه هستند بنابراین دو عنصر [E→.a,$+] و [E→.a,+$] یکسان هستند.

[S→.E-E,$]- در closure([S→.E,$]) و E→a در گرامر و first(-E$)={-} است بنابراین [E→.a,-] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$], [E→.E+E,$] , [E→.E-E,$] , [E→.a,$+], [E→.a,-] }

نمادهای پیشگویی دو عنصر [E→.a,$+],[E→.a,-] را ترکیب می‌کنیم. بنابراین عنصر [E→.a,$+-] را به مجموعه اضافه می‌کنیم. درنتیجه :

closure([S→.E,$])={[S→.E,$], [E→.E+E,$] , [E→.E-E,$] , [E→.a,$+-] }

[S→.E+E,$]- در closure([S→.E,$]) و E→E+E در گرامر و first(+E$)={+} است بنابراین [E→.E+E,+] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$], [E→.E+E,$] , [E→.E-E,$] , [E→.a,$+-], [E→.E+E,+] }

نمادهای پیشگویی دو عنصر [E→.E+E,$], [E→.E+E,+] را ترکیب می‌کنیم. بنابراین عنصر [E→.E+E,$+] را به مجموعه اضافه می‌کنیم. درنتیجه :

closure([S→.E,$])={[S→.E,$], [E→.E+E,$+] , [E→.E-E,$] , [E→.a,$+-] }

[S→.E-E,$]- در closure([S→.E,$]) و E→E+E در گرامر و first(-E$)={-} است بنابراین [E→.E+E,-] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$],[E→.E+E,$+] ,[E→.E-E,$] ,[E→.a,$+-],[E→.E+E,-] }

نمادهای پیشگویی دو عنصر [E→.E+E,$+], [E→.E+E,-] را ترکیب می‌کنیم. بنابراین عنصر [E→.E+E,$+-] را به مجموعه اضافه می‌کنیم. درنتیجه :

closure([S→.E,$])={[S→.E,$], [E→.E+E,$+-] , [E→.E-E,$] , [E→.a,$+-] }

[S→.E+E,$]- در closure([S→.E,$]) و E→E-E در گرامر و first(+E$)={+} است بنابراین [E→.E-E,+] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$], [E→.E+E,$+-] , [E→.E-E,$] , [E→.a,$+-], [E→.E-E,+] }

نمادهای پیشگویی دو عنصر [E→.E-E,$], [E→.E-E,+] را ترکیب می‌کنیم. بنابراین عنصر [E→.E-E,$+] را به مجموعه اضافه می‌کنیم. در نتیجه :

closure([S→.E,$])={[S→.E,$], [E→.E+E,$+-] , [E→.E-E,$+] , [E→.a,$+-] }

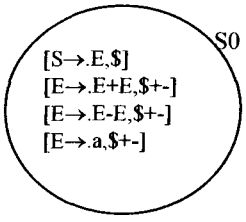
[S→.E-E,$]- در closure([S→.E,$]) و E→E-E در گرامر و first(-E$)={-} است بنابراین [E→.E-E,-] به مجموعه closure([S→.E,$]) اضافه می‌شود. در نتیجه:

closure([S→.E,$])={[S→.E,$], [E→.E+E,$+-] , [E→.E-E,$+], [E→.a,$+-] , [E→.E-E,-] }

نمادهای پیشگویی دو عنصر [E→.E-E,$+], [E→.E-E,-] را ترکیب می‌کنیم. بنابراین عنصر [E→.E-E,$+-] را به مجموعه اضافه می‌کنیم. درنتیجه :

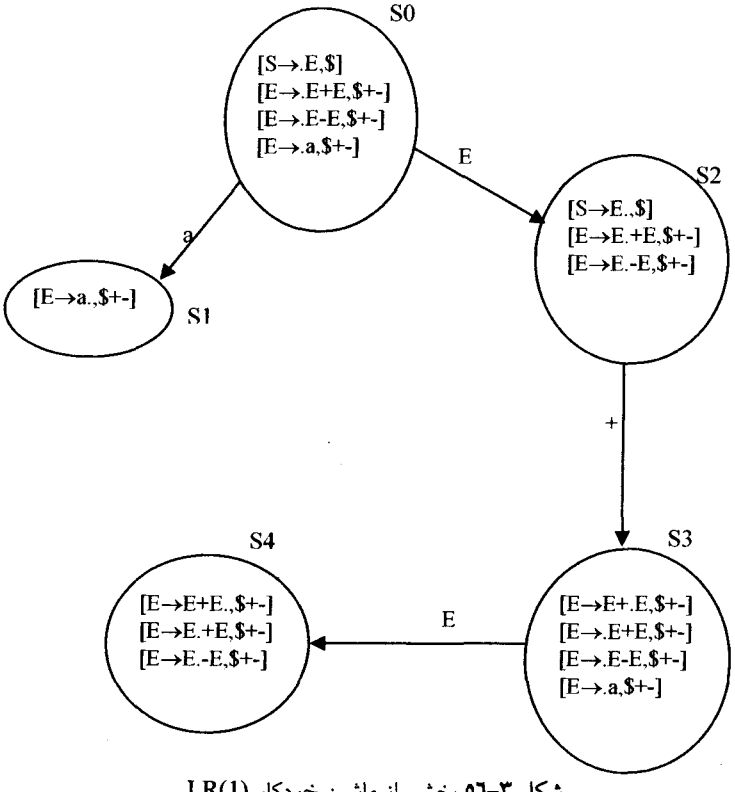
closure([S→.E,$])={[S→.E,$], [E→.E+E,$+-] , [E→.E-E,$+-] , [E→.a,$+-] }

این مجموعه را S0 می‌نامیم.



شکل ۳-۵۵ بخشی از ماشین خودکار LR(1)

حالات دیگر را به روش ذکر شده محاسبه و ماشین خودکار را رسم می‌کنیم. بخشی از ماشین خودکار در ذیل ارائه شده است.



شکل ۳-٥٦ بخشی از ماشین خودکار LR(1)

حالت S4 شامل عنصر کاهشی [E→E+E.,$+-] و عنصر انتقالی [E→E.+E,$+-] است. اگر نماد ورودی + باشد، امکان کاهش [E→E+E.,$+-] و انتقال [E→E.+E,$+-] است، این وضعیت نشان دهنده برخورد انتقال/کاهش است در نتیجه گرامر LR(1) نیست.

تعداد حالاتی که در LR(1) تولید می‌گردد بیشتر از حالات تولید شده به روشهای LR(0) و SLR(1) است. در نتیجه در عمل کمتر از این روش استفاده می‌گردد. LR(1) از همه روشها قوی‌تر است در نتیجه اگر گرامری LR(1) نباشد، LR(0)،SLR(1) نیز نیست. به عنوان نمونه گرامر مثال۳-۷۷ LR(1) نیست در نتیجه LR(0) و SLR(1) نیز نیست. همچنین اگر گرامری LR(0) و SLR(1) باشد، LR(1) نیز می‌باشد.

۳-۱۸-٤ روش LALR(1)[1]

روشهای LR(0) و SLR(1) ضعیف هستند، یعنی برای تعداد کمی از گرامرها می‌توان به این روش جدول تجزیه ساخت. روش LR(1) به اندازه کافی قوی است تا بتوان برای بسیاری از گرامرها از جمله زبانهای برنامه نویسی جدول تجزیه آنها را تولید کرد. اما حالات LR(1) خیلی زیاد است که خود مشکل بزرگی برای ساخت جدول تجزیه است. روش LALR(1)، که در این بخش به توضیح آن خواهیم پرداخت یک راه حل میانه است. LALR(1) از LR(0) و SLR(1) قویتر و از LR(1) ضعیفتر است. با اینکه LALR(1) از LR(1) ضعیف‌تر است، اما هنوز قادر به تولید تجزیه کننده برای بسیاری از گرامرهای پر استفاده به ویژه زبانهای برنامه نویسی است. LALR(1) نسبت به LR(1) تعداد حالات کمتری را ایجاد می‌کند به همین جهت استفاده از آن به ویژه برای زبانهای برنامه نویسی عملی‌تر است.

برای تشریح LALR(1)، ابتدا حالت هسته[2] را تعریف می‌کنیم. حالت هسته، مجموعه ای از عناصر پیشگویی است که پایانه‌های پیشگویی آن حذف شده است. به عنوان مثال حالتS3 را در نظر می‌گیریم. حالت هسته معادل آن به صورت ذیل است.

جدول ۳-۵۴ جدول تجزیه LALR(1)

حالت هسته LR(1)	حالت S3 ماشین خودکار از LR(1)
E→ d.Eb E→ .dEb E→.T T→.a	[E→ d.Eb ,$] [E→ .dEb,b] [E→.T,b] [T→.a,b]

1. Look Ahead LR
2. Core

هسته‌های مربوط به حالتهای LR(1) معادل حالتهای LR(0) است. در واقع LR(1) هر یک حالتهای LR(0) را به چند حالت شکسته است، و این شکستن‌ها منشا قدرت LR(1) است. ولی این شکستن‌ها در برخی حالات نیاز نیست.

با بررسی مجموعه حالات LR(1) مشخص می‌گردد برخی از هسته‌ها تکرار شده اند. می‌توان با ترکیب این حالات ، تعداد حالات را کم کرد. در هنگام ترکیب حالات هسته مجموعه پایانه‌های پیشگویی را نیز با یکدیگر جمع می‌کنیم. به عنوان مثال ماشین خودکار شکل ۳-۵۴ را بررسی می‌کنیم. در جدول ۳-۵۵ حالات هسته تکراری ماشین خودکار و ترکیب آنها ارائه شده است.

جدول ۳-۵۵ حالات ترکیبی

حالت هسته قابل ترکیب اول	حالت هسته قابل ترکیب دوم	حالت نتیجه با جمع پایانه‌های پیشگویی
S8:[E→ dEb. ,$]	S12:[E→ dEb. ,b]	S8,12[E→ dEb. ,b$]
S4:[E→T. , $]	S9: [E→T. , b]	S4,9: [E→T. , b$]
S6:[E→ dE.b,$]	S11: [E→ dE.b,b]	S6,11: [E→ dE.b,$b]
S3: [E→ d.Eb ,$] [E→ .dEb ,b] [E→.T , b] [T→.a , b]	S10: [E→ d.Eb ,b] [E→ .dEb ,b] [E→.T , b] [T→.a , b]	s3,10: [E→ d.Eb ,b$] [E→ .dEb ,b] [E→.T , b] [T→.a , b]

با بررسی بیشتر ماشین خودکار LR(1) حالات مشابه دیگری یافت نمی‌شود. حالات ترکیب شده را به صورت یک حالت جدید با نام هر دو حالت ترکیب شده در نظر می‌گیریم. یعنی اگر حالت 6 با حالت 11 ترکیب شود، حالت ترکیبی را S11,6 می‌نامیم.

ماشین خودکار LALR(1) شامل دو دسته حالات است، که عبارتند از:

۱- حالات غیر ترکیبی: حالاتی از ماشین خودکار LR(1) که ترکیب نشده اند. مانند حالات: S0,S1,S2,S5,S7,S13

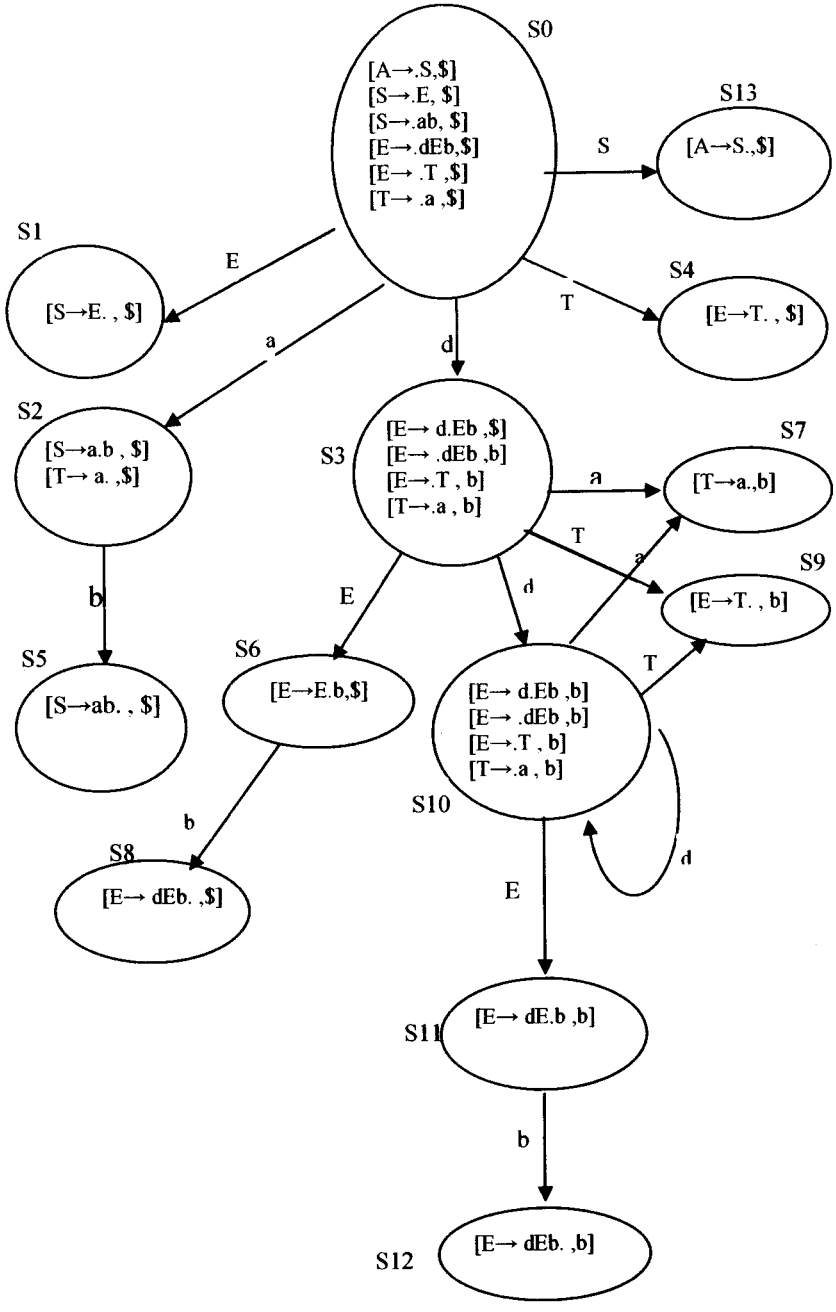
۲- حالات ترکیبی: حالاتی که از ترکیب حالات LR(1) به دست آمده اند. مانند:

S3,10: ترکیب حالات S3 و S10

S6,11: ترکیب حالات S11 و S6

S4,9: ترکیب حالات S9 و S4

S8,12: ترکیب حالات S12 و S8



شکل ۳-۵۷ ماشین خودکار LR(1)

اگر در ماشین خودکار LR(1) تغییر حالتی از Si به Sj وجود دارد در این صورت در ماشین خودکار LALR(1) نیز تغییر حالتی از حالت شامل Si به حالت شامل Sj وجود خواهد داشت.

جدول ۳-۵۶ تناظر تغییر حالات LR(1) با LALR(1)

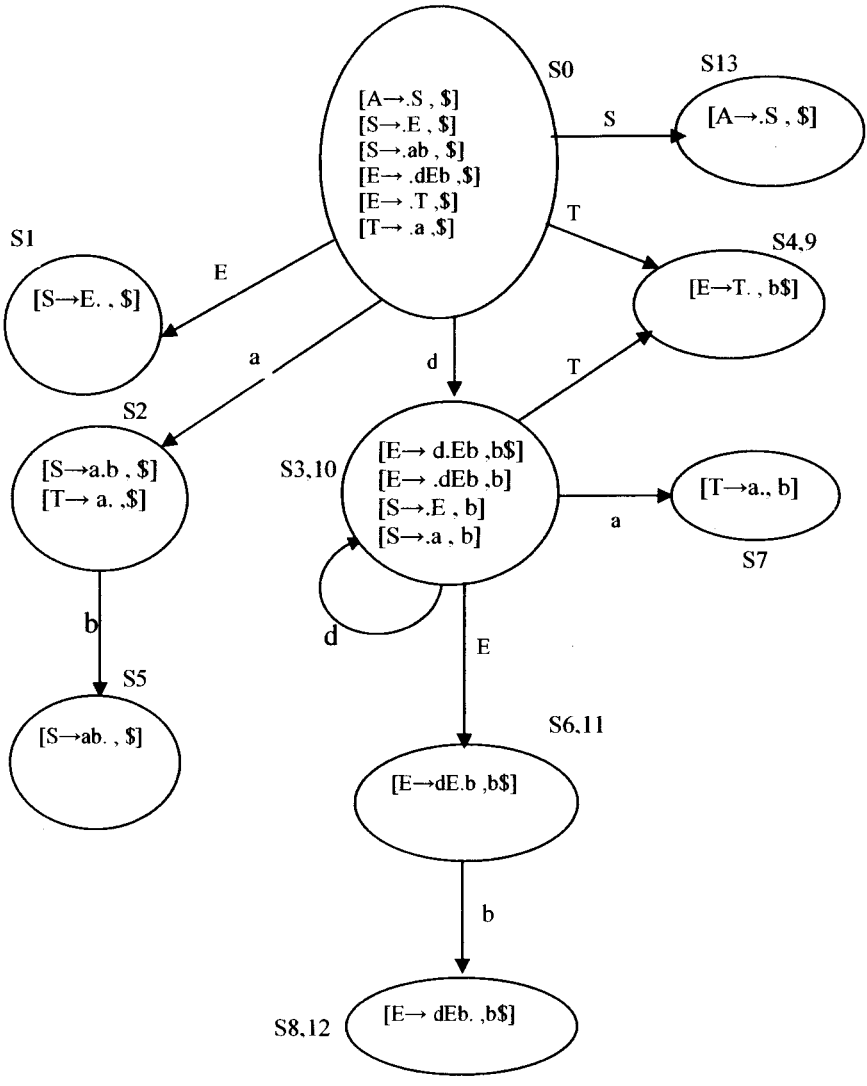
تغییر حالت ماشین خودکار LR(1)	تغییر حالت ماشین خودکار LALR(1)
از S0 به S3	از S0 به S3,10
از S3 به S6	از S3,10 به S6,11
از S3 به S7	از s3,10 به S7
...	...

نتیجه ایجاد ماشین خودکار LALR(1) در شکل ۳-۵۸ نشان داده شده است. نحوه ساخت جدول تجزیه LALR(1) از ماشین خودکار مانند LR(1) است. حال با استفاده از ماشین خودکار LALR(1) می‌توان جدول تجزیه را به صورت ذیل ایجاد کرد.

جدول ۳-۵۷ جدول تجزیه LALR(1)

حالات	action				goto		
	a	b	d	$	E	T	S
0	s2		s3,10		1	4,9	13
1	r2			r2			
2		s5		r6			
3,10	s7		s3,10		6,11	4,9	
4,9		r5		r5			
5				r3			
6,11		s8,12					
7		r6					
8,12				r4			
13				accept			

همانطور که ملاحظه می‌گردد جدول LALR(1) نسبت به LR(1) تعداد حالات و در نتیجه جدول تجزیه کوچکتری دارد. در نتیجه پیاده سازی این روش از LR(1) مناسب‌تر است.



شکل ۳-۵۸ ماشین خودکار LALR(1)

با توجه به گرامر ذیل و جدول تجزیه ۳-۵۷ رشته $dab را تجزیه می‌کنیم.

1- A→ S
2- S→ E
3- S→ ab
4- E→ dEb
5- E→ T
6- T→ a

جدول ۳-۵۸ تجزیه رشته dab

پشته	رشته ورودی
0	dab$
0d3,10	ab$
0d3,10a7	b$
0d3,10a7	b$
0d3,10T	b$
0d3,10T4,9	b$
0d3,10T4,9	b$
0d3,10E	b$
0d3,10E6,11	b$
0d3,10E6,11b8,12	$
0E	$
0E1	$
0S	$
0S13 (accept)	$

اگر ترکیب حالات موجب برخورد انتقال/کاهش گردد، گرامر مورد نظر LALR(1) نیست.

مثال۳-۷۸ به گرامر ذیل توجه کنید.

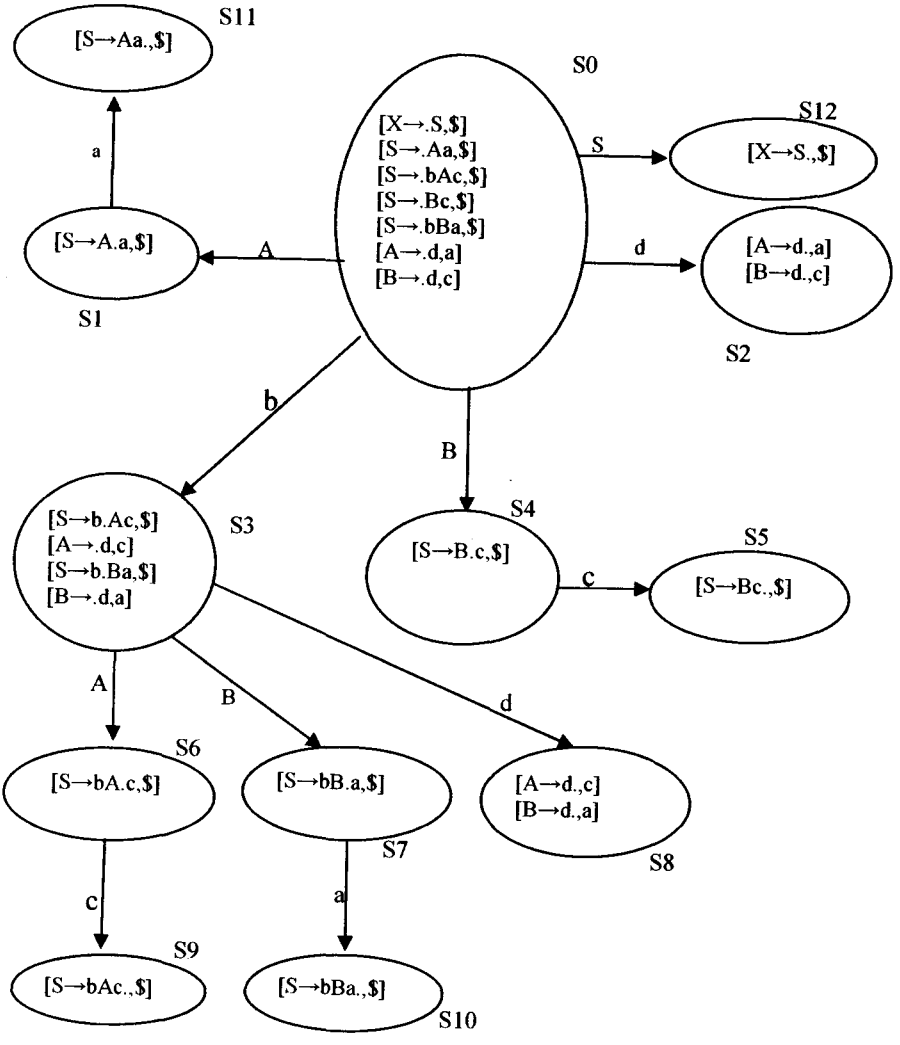
S→ Aa | bAc | Bc | bBa
A→d
B→d

گرامر فوق را به صورت ذیل تغییر می‌دهیم

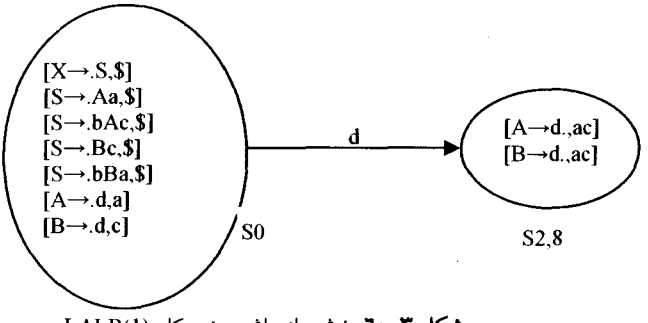
1- X→ S
2- S→ Aa
3- S→ bAc
4- S→ Bc
5- S→ bBa
6- A→d
7- B→d

برای ایجاد ماشین خودکار LALR(1) ابتدا ماشین خودکار LR(1) را باید تولید کنیم. ماشین خودکار LR(1) گرامر در شکل ۳-۵۹ رسم شده است.

بررسی ماشین خودکار ۳-۵۹ نشان می‌دهد که حالات S2 و S8 دارای هسته‌های یکسانی هستند در نتیجه طبق روش LALR(1) آنها را ترکیب می‌کنیم. نتیجه ترکیب حالات در شکل ۳-۶۰ نشان داده شده است.



شکل ۳-۵۹ ماشین خودکار LR(1)



شکل ۳-٦۰ بخشی از ماشین خودکار LALR(1)

با توجه به شکل فوق در S2,8 برخورد کاهش/کاهش رخ داده است. زیرا پایانه‌های پیشگویی a و c بین عناصر [A→d.,ac] و[B→d.,ac] مشترک است، در نتیجه اگر در این حالت نماد ورودی a باشد مشخص نیست کدام یک از کاهش‌ها باید انجام شود. در نتیجه از روی عناصر پیشگویی نیز قادر به برطرف کردن برخورد نیستیم. بنابراین این گرامر LALR(1) نیست ولی LR(1) است.

ساخت ماشین خودکار LALR(1) نمی‌تواند موجب برخورد انتقال/کاهش شود. برای اثبات این مسئله، فرض می‌کنیم یک ماشین خودکار LR(1) موجود است و ماشین خودکار LALR(1) را از آن تولید می‌کنیم. برخورد انتقال/ کاهش زمانی رخ می‌دهد که یک عنصر کاهشی مانند [S→α.,a] و یک عنصر انتقالی مانند [B→β.aρ,b] در یک حالت LALR(1) وجود داشته باشد. چون عناصر حالات LALR(1) از ماشین خودکار LR(1) به دست می‌آید. در نتیجه الزاما می بایست حداقل دو عنصر [S→α.,a] و [B→β.aρ,c] در یکی از حالات ماشین خودکار LR(1) موجود باشد. اگر چنین باشد ماشین خودکار LR(1) دچار برخورد انتقال/کاهش است و در نتیجه LR(1) نیست که این با فرض اول در تناقض است، زیرا ابتدا فرض کردیم ماشین خودکار اولیه LR(1) است.

تجزیه کننده‌های LR وقتی خطا را تشخیص می‌دهد که در جدول تجزیه در قسمت action ، error (یا خالی باشد) را بیابد، تجزیه کننده‌های LR خطا را با استفاده از goto تشخیص نمی‌دهند در واقع تجزیه کننده فقط در قسمت action با error مواجه می گردد. تجزیه کننده LR(1) سریعتر از روشهای دیگر خطا را کشف و گزارش می‌کند. اما تجزیه کننده‌های SLR(1) و LALR(1) ممکن است چند کاهش را انجام دهند تا خطا را کشف کنند.

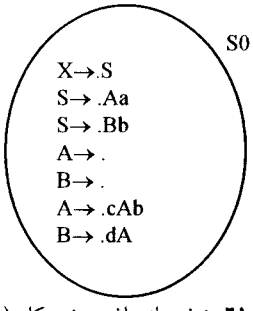
LR(1) از همه روشها قوی تر است. بنابراین اگر گرامری LR(1) نباشد LALR(1) نیز نیست. چون LALR(1) از SLR(1) و LR(0) قوی تر است. اگر گرامری LR(0) و SLR(1) باشد، LALR(1) نیز می‌باشد و اگر گرامری LALR(1) نباشد، LR(0) وSLR(1) نیز نیست.

گرامرهای مبهم، LALR(1),SLR(1),LR(0) و LR(1) نیستند. به عنوان نمونه گرامر مثال ۳-۷۷ مبهم است در نتیجه این گرامرLALR(1),LR(1)،SLR(1)،LR(1) نیست. بنابراین اگر گرامری LALR(1),SLR(1),LR(1)و یا LR(0) باشد، ثابت می‌شود که گرامر مبهم نیست. بنابراین برای اثبات غیر مبهم بودن یک گرامر می‌توان اثبات کرد گرامر یکی از انوان گرامرهای LR است. اما عکس این مطلب صحیح نیست یعنی ممکن است گرامر غیر مبهمی وجود داشته باشد که LALR(1),SLR(1),LR(0) ویا LR(1) نباشد.

مثال۳-۷۹ SLR(1) و LR(1) و LALR(1) بودن گرامر ذیل را بررسی کنید.

S→ Aa
S→ Bb
A→ ∈
B→ ∈
A→ cAb
B→ dAa

به منظور بررسی SLR(1) بودن گرامر، ماشین خودکار آن را به روش SLR(1) رسم می‌کنیم. ابتدا به گرامر قاعده تولید X→S را اضافه کرده و قواعد را شماره گذاری می‌کنیم. در ذیل بخشی از ماشین خودکار گرامر ارائه شده است.

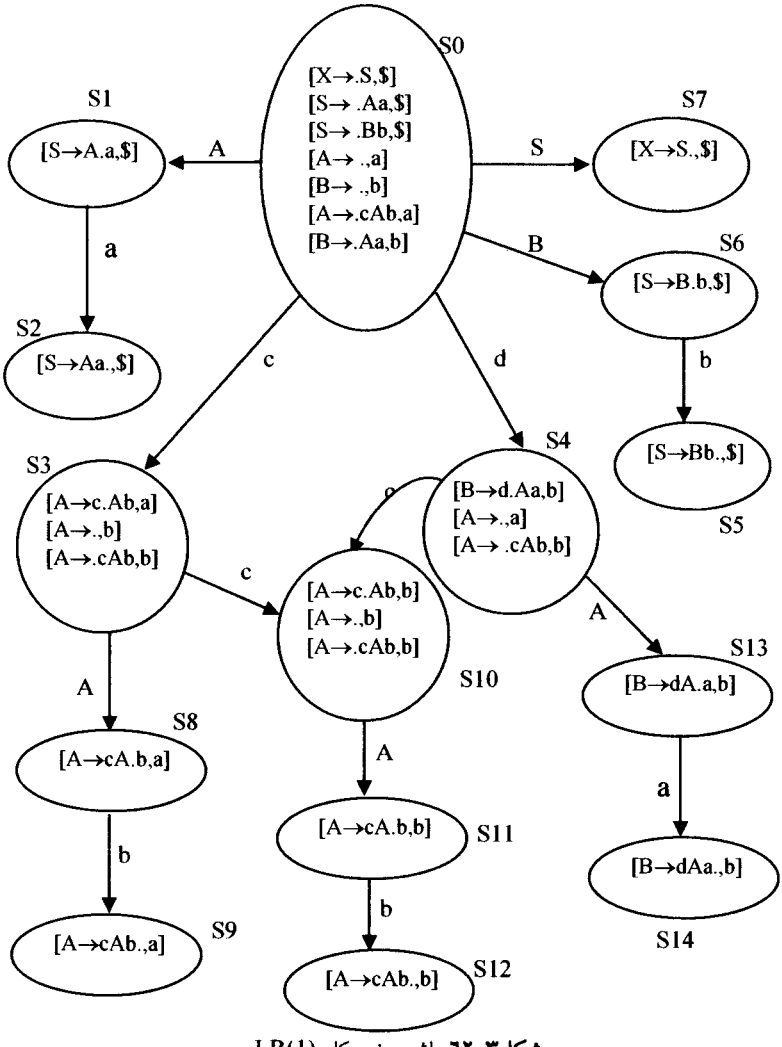


شکل ۳-۶۱ بخشی از ماشین خودکار SLR(1)

حالت S0 شامل عنصر کاهشی .A→ است، کاهش A برای پایانه‌های follow(A)={a,b} مجاز است. همچنین S0 شامل عنصر کاهشی .B→ نیز می‌باشد که کاهش B برای پایانه‌های follow(B)={b} نیز مجاز است. در نتیجه اگر نماد ورودی در S0، b باشد کاهش .A→ و

.B→ هر دو مجاز است که این وضعیت برخورد کاهش/کاهش است و در نتیجه گرامر SLR(1) نیست.

برای بررسی LR(1) بودن ابتدا ماشین خودکار LR(1) رسم می‌کنیم.



شکل۳-۶۲ ماشین خودکار LR(1)

با بررسی ماشین خودکار هیچ نوع برخوردی کشف نمی‌شود در نتیجه گرامر LR(1) است با توجه به اینکه گرامر LR(1) است LALR(1) گرامر را بررسی می‌کنیم، برای بررسی LALR(1)

بودن گرامر با توجه به ماشین خودکار حالاتی که هسته یکسان دارند را ترکیب می‌کنیم. جدول ذیل حالات قابل ترکیب را نشان می‌دهد.

جدول۳-۵۹ حالات ترکیبی

حالت اول	حالت دوم	حالت ترکیب
S8:[A→cA.b,a]	S11:[A→cA.b,b]	S8,11:[A→cA.b,ab]
S9:[A→cAb.,a]	S12:[A→cAb.,b]	S9,12:[A→cAb.,ab]
S3:[A→c.Ab,a] [A→.,b] [A→.cAb,b]	S10:[A→c.Ab,b] [A→.,b] [A→.cAb,b]	S3,10:[A→c.Ab,ab] [A→.,b] [A→.cAb,b]

حالات ترکیبی به دست آمده نشان دهنده برخورد کاهش/کاهش یا انتقال/کاهش نیستند در نتیجه گرامر LALR(1) است.

۳-۱۹- گرامرهای مبهم

علی رغم قدرت زیاد تجزیه کننده‌های LR، برای گرامرهای مبهم نمی‌توان تجزیه کننده LR ایجاد کرد، به عبارت دیگر گرامرهای مبهم، گرامرLR نیستند. دو راهکار وجود دارد که عبارتند از:

۱- تبدیل گرامر مبهم به گرامر غیر مبهم معادل آن

۲- استفاده از قوانین اضافی

در ذیل به بررسی هر یک از این راهکارها می‌پردازیم.

- تبدیل گرامر مبهم به گرامر غیر مبهم

در بسیاری از موارد تبدیل یک گرامر مبهم به گرامر غیر مبهم بسیار مشکل است، همچنین گرامرهایی وجود دارند که معادل غیر مبهم ندارند این گونه گرامرها را ذاتا مبهم می‌نامیم. در برخی موارد گرامرهای مبهم ، ساختار زبان مبدا را قابل فهم تر بیان می‌کنند. نکته مهم دیگر آن است که گرامرهای مبهم تعداد و قواعد تولید کمتری نسبت به گرامر غیر مبهم معادل خود دارند. به عنوان مثال به گرامر مبهم ذیل دقت کنید:

E→E + E | E * E | (E) | id

این گرامر مبهم شامل ٤ قاعده تولید و یک غیر پایانه است همچنین درک این گرامر برای سازندگان زبان نیز ساده است. به گرامر غیر مبهم معادل آن که در ذیل داده شده است توجه کنید.

E→ E + T | T
T→ T * F | F

F→ (E) | id

این گرامر غیر مبهم شامل ٦ قاعده تولید سه غیر پایانه است. تعداد قواعد تولید و غیر پایانه‌ها روی جدول تجزیه تاثیر می‌گذارد . با توجه به مشکلات مربوط به تبدیل یک گرامر مبهم به گرامر غیر مبهم، و تاثیر گرامر غیر مبهم معادل روی جدول تجزیه روش دوم یعنی تغییر تجزیه کننده را بررسی می‌کنیم.

- استفاده از قوانین

دو برخورد انتقال/کاهش و برخورد کاهش/کاهش علت شکست در تولید تجزیه کننده LR است. اگر به نحوی این برخوردها را رفع کنیم می‌توانیم تجزیه کننده LR را بسازیم. برخورد انتقال/کاهش ، زمانی رخ می‌دهد که تجزیه کننده در یک حالت خاص امکان انتقال و کاهش را داشته باشد. می‌توان توسط قوانین اضافی که به تجزیه کننده اعمال می‌گردد، همواره یکی انتخاب گردد، در این صورت برخورد به وجود نمی‌آید.

از جمله روشهایی که برای رفع برخورد به کار می‌رود عبارتند از :

۱- رفع برخورد انتقال/کاهش: سعی می‌کنیم طولانی ترین دنباله را بیابیم به همین جهت در برخورد انتقال/کاهش، عمل انتقال انجام می‌گردد.

روش دیگر برای رفع برخورد انتقال/کاهش، استفاده از تقدم (یا اولویت بندی بین نمادها) و شرکت پذیری است.

۲- رفع برخورد کاهش/کاهش : در برخورد کاهش/کاهش، قاعده تولیدی انتخاب می‌شود که طولانی ترین رشته را کاهش دهد. اگر چند قاعده تولید وجود داشته باشد که طول یکسانی از رشته را می‌پذیرند، در این صورت از اولویت بندی بین قواعد تولید استفاده می‌شود. در این صورت گرامری که اول نوشته شده است ، اولویت دارد

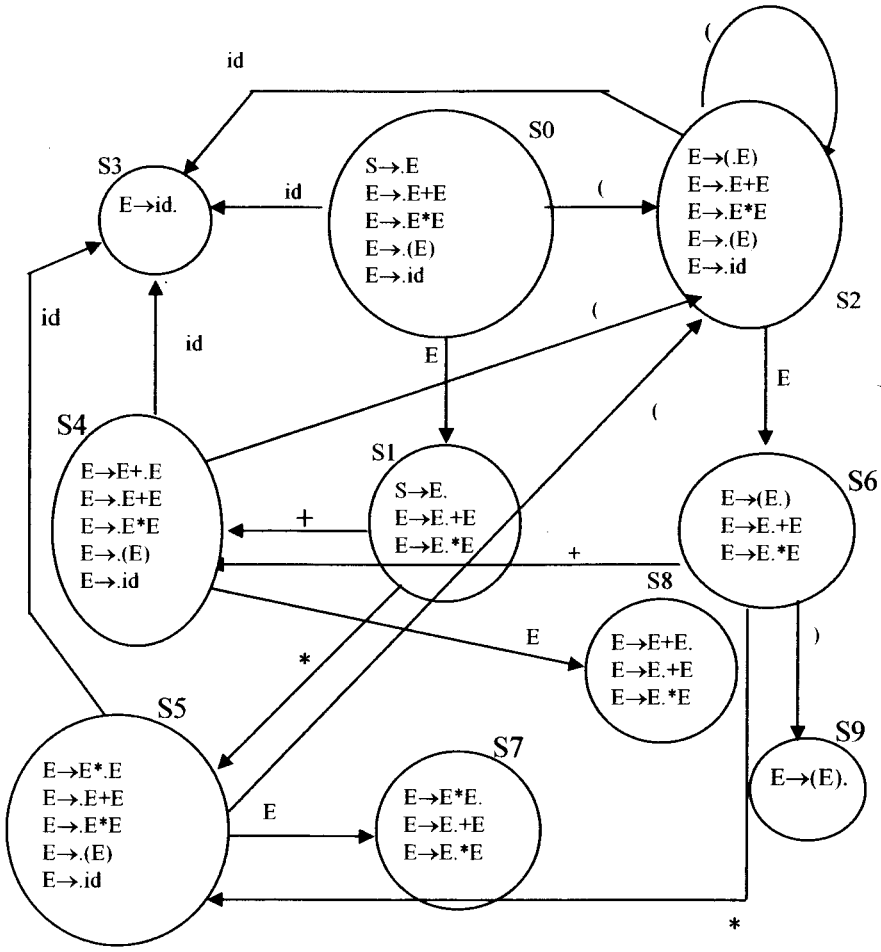
مثال۳-۸۰ گرامر ذیل را در نظر می‌گیریم.

E→ E + E | E * E | (E) | id

ابتدا به گرامر یک قاعده تولید اضافه کرده و قواعد تولید را به صورت ذیل شماره گذاری می‌کنیم.

1- S→E
2- E→E * E
3- E→E + E
4- E→(E)
5- E→ id

برای این گرامر بخشی از ماشین خودکار تجزیه کننده SLR(1) به صورت ذیل تولید می‌گردد.



شکل ۳-۶۳ ماشین خودکار SLR(1)

همانطور که ملاحظه می‌گردد این گرامر SLR(1) نیست، زیرا در حالات S7 و S8 برخورد انتقال/کاهش وجود دارد. زیرا امکان انتقال و کاهش وجود دارد. در نتیجه باید به روشی برخورد را رفع کنیم، یعنی باید بین انتقال و کاهش یکی را انتخاب کنیم. انتخاب یکی از این دو بستگی به شرایط زبان دارد. به عنوان نمونه در مثال ذکر شده انتخاب یکی از این دو بستگی به اولویت یا تقدم عملگرها دارد. اگر اولویت * نسبت به + بیشتر باشد و + و * شرکت پذیری چپ داشته باشند، رفع برخورد می‌تواند به صورت ذیل باشد:

- در حالت S8 برای رفع برخورد انتقال/کاهش، کاهش انتخاب می‌گردد.

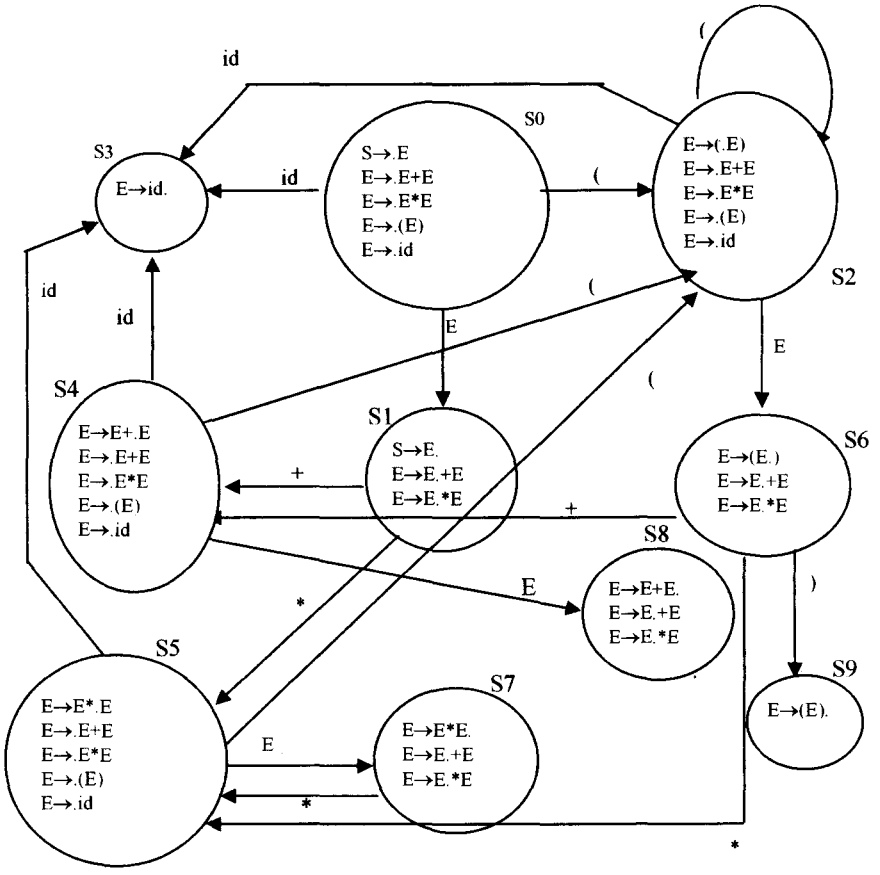
- حالت S7 اگر نماد ورودی * باشد، انتقال و اگر نماد ورودی + باشد کاهش انتخاب می‌گردد.

با توجه به رفع برخوردهای ارائه شده می‌توان ماشین خودکار SLR(1) را به صورت شکل ۳-۶۴ نشان داد.

با توجه به رفع برخوردهای ذکر شده جدول تجزیه به صورت ذیل خواهد شد.

جدول ۳-۶۰ جدول تجزیه

حالت	Action						goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			8
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	



شکل ۳-٦٤ ماشین خودکار SLR(1)

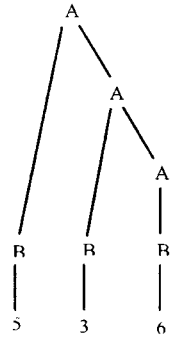
۳-۲۰ ترجمه

کامپایلر یک رشته ورودی (به عنوان مثال برنامه به زبان پاسکال) را به رشته خروجی (برنامه ای به زبان اسمبلی) ترجمه می‌کند. فرایند ترجمه برای تبدیل رشته ورودی به رشته خروجی می‌تواند در ضمن تجزیه انجام شود. تجزیه یک روند مرحله به مرحله است اگر متناسب با هر مرحله از فرایند تجزیه، ترجمه مورد نیاز برای آن مرحله انجام شود، ترجمه رشته ورودی پس از اتمام تجزیه به دست می‌آید. تجزیه کننده‌ها برای رشته ورودی، درخت تجزیه را مرحله به مرحله می‌سازند ، اگر عمل ترجمه همزمان با ساخت درخت تجزیه انجام شود، در این صورت پس از پایان ساخت درخت تجزیه می‌توان ترجمه را نیز تکمیل کرد (به شرط آنکه برنامه مبدا صحیح باشد).

هر گره از درخت تجزیه مطابق یکی از قواعد تولید گرامر است. در نتیجه، برای هر قاعده تولید، ترجمه را مشخص می‌کنیم. به عنوان مثال فرض کنید می‌خواهیم ، دنباله‌ای شامل اعداد مبنای هشت را به معادل دودویی آن ترجمه کنیم. به منظور ترجمه اعداد مبنای هشت به دودویی، هر رقم را جداگانه به دودویی ترجمه می‌کنیم. گرامر تولید یک عدد در مبنای هشت را به صورت ذیل است.

A→ BA | B
B→ 0 |1 | 2 | 3 | 4 | 5 | 6 |7

درخت تجزیه برای رشته 536 گرامر به صورت ذیل است.



شکل ۳-۶۵ درخت تجزیه رشته 536

اگر برگهای درخت تجزیه را از چپ به راست بخوانیم رشته 536 به دست می‌آید. برای هر یک از قواعد تولید، ترجمه ای به صورت ذیل در نظر می‌گیریم.

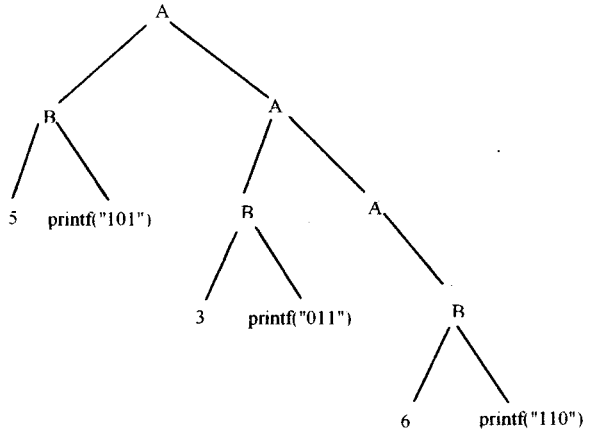
جدول ۳-۶۱ قوانین ترجمه

قاعده تولید	ترجمه(خروجی)
B→ 0	000
B→ 1	001
B→ 2	010
B→ 3	011
B→ 4	100
B→ 5	101
B→ 6	110
B→ 7	111

این جدول نشان می‌دهد هرگاه B→7 کاهش یابد، رقم 7 در رشته ورودی دیده شده است که در این صورت ترجمه آن 111 است. قوانین ترجمه در هنگام کاهش به غیر پایانه‌ها اجرا می‌شوند. می‌توان قوانین ترجمه را در قواعد تولید گنجاند. گرامر و ترجمه را می‌توان به صورت ذیل نشان داد.

```
A→ BA
A→ B
B→ 0   { printf("000")}
   | 1   { printf("001")}
   | 2   { printf("010")}
   | 3   { printf("011")}
   | 4   { printf("100")}
   | 5   { printf("101")}
   | 6   { printf("011")}
   | 7   { printf("111")}
```

در این گرامر دستورات ترجمه جزئی از گرامر است. درخت تجزیه برای رشته 536 با توجه به گرامر و دستورات ترجمه جدید درخت تجزیه به صورت ذیل است.



شکل ۳-٦٦ درخت تجزیه 536 به همراه ترجمه آن

دستورات ترجمه موجود در درخت جدید را به ترتیبی که در پیمایش inorder دیده می‌شوند اجرا می‌شوند. در نتیجه دستورات به ترتیب ذیل اجرا می‌گردند.

```
printf("101")
printf("011")
printf("110")
```

حاصل اجرای این دستورات رشته 101011110 است که ترجمه دودویی رشته 536 است. به عنوان مثال دیگر به گرامر ذیل دقت کنید.

```
A→ B T
T → + B T | - B T | ∈
B→1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

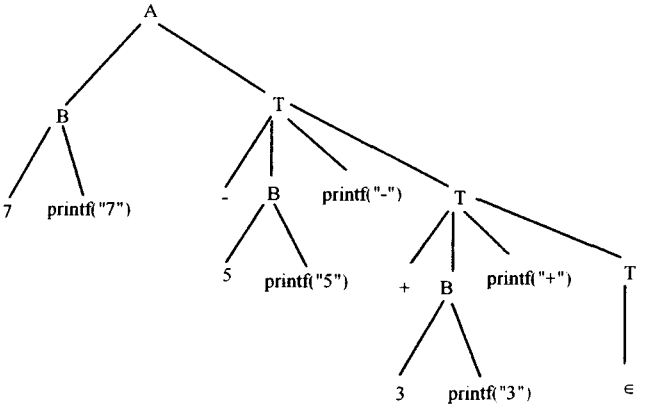
این گرامر عبارات محاسباتی یک رقمی تولید می‌کند. برای ترجمه عبارات به معادل پسوندی از دستورات ترجمه ذیل استفاده می‌گردد.

```
A→ B T
T → + B { printf("+");} T | - B { printf("+");}T | ∈
```

```
B→1  { printf("1");}
B→2  { printf("2");}
B→3  { printf("3");}
B→4  { printf("4");}
B→5  { printf("5");}
B→6  { printf("6");}
B→7  { printf("7");}
B→8  { printf("8");}
B→9  { printf("9");}
```

دستورات ترجمه مانند پایانه و غیر پایانه‌ها در قواعد تولید قرار می‌گیرند. مکان این دستورات در درخت تجزیه نیز بر اساس همین ترتیب مشخص می‌گردد.

مثال۳-۸۱ درخت تجزیه برای رشته 3+5-7 به صورت ذیل است.



شکل ۳-٦۷ درخت تجزیه با دستورات ترجمه

اگر دستورات ترجمه را به ترتیب پیمایش inorder در نظر بگیریم ترتیب ذیل به دست می‌آید.

```
printf("7");
printf("5");
printf("-");
printf("3");
printf("+");
```

اجرای این دستورات رشته +3-75 را چاپ می‌کند. رشته +3-75 معادل پسوندی رشته3+5-7 است. به عبارت دیگر رشته 3+5-7 به رشته +3-75 ترجمه شده است. اینکه چه قوانینی را در کدام قسمت گرامر باید قرار دهیم، الگوریتم مشخصی ندارد، بلکه برنامه نویس با توجه به رشته ورودی ، گرامر و آنچه باید تولید گردد، باید دستورات صحیح و مکان آنها را تشخیص دهد. اگر گرامر تغییر کند نوع دستورات و محل آنها نیز تغییر خواهد کرد. به عنوان مثال فرض کنید برای تولید عبارات محاسباتی از گرامر ذیل استفاده گردد.

A→A OP B | B
OP→ + | -
B→1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

در این صورت از دستورات ترجمه دیگری باید استفاده گردد. در مثالهای قبلی ترجمه یک گره همزمان با کاهش آن چاپ می‌شد. در برخی موارد ترجمه هر قسمت را می‌خواهیم نگهداری کنیم تا در موقع مناسب چاپ یا برای روال دیگری ارسال کنیم. در ترجمه می‌توان برای گره‌های درخت تجزیه صفاتی را در نظر گرفت. به عنوان مثال گرامر ذیل را در نظر می‌گیریم.

A→A OP B| B
OP→ + | -
B→1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

این گرامر رشته‌های محاسباتی میانوندی را تولید می‌کند. می‌خواهیم رشته‌های محاسباتی را به معادل پسوندی آن ترجمه کنیم. برای ترجمه رشته‌های محاسباتی با توجه به گرامر نیاز به جمع آوری اطلاعات از گره‌های پایین و استفاده از آن در گره‌های بالاتر است. بدین جهت می‌توان در گرامر برای هر غیر پایانه به تعداد مورد نیاز صفت در نظر گرفت. با استفاده از مقادیر صفات گره‌های فرزند، مقادیر صفات گره پدر را محاسبه می‌کنیم.

به عنوان مثال برای هر غیر پایانه در گرامر مورد صفت x را در نظر می‌گیریم. صفت x در هر غیر پایانه معادل پسوندی فرزندان آن گره را نگهداری می‌کند. برای محاسبه صفتx یک غیر پایانه با استفاده از صفت x فرزندان آن از قوانینی به صورت ذیل استفاده می‌شود (علامت || به معنی اتصال دو رشته است.).

جدول ۳-۶۲ قوانین ترجمه

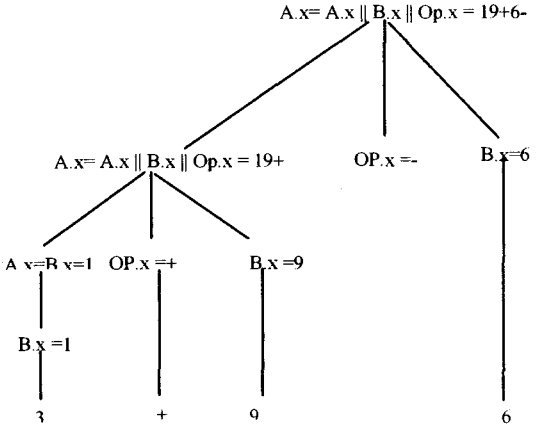
قاعده تولید	قانون ترجمه
A→A OP B	A.x= A.x || B.x || OP.x
A→ B	A.x=B.x
OP→ +	OP.x='+'
OP→ -	OP.x='-'
B→1	B.x='1'
B→ 2	B.x='2'
B→ 3	B.x='3'
B→ 4	B.x='4'
B→ 5	B.x='5'
B→ 6	B.x='6'
B→ 7	B.x='7'
B→ 8	B.x='8'
B→ 9	B.x='9'

با توجه به قوانین ترجمه گرامر به صورت ذیل تغییر می‌کند.

A→A OP B | B { A.x= A.x || B.x || OP.x}
A→ B { A.x= B.x }

OP→ + { OP.x='+'}
OP→ - { OP.x='-'}
B→1 {B.x=1}
B→2 {B.x=2}
B→3 {B.x=3}
B→4 {B.x=4}
B→5 {B.x=5}
B→6 {B.x=6}
B→7 {B.x=7}
B→8 {B.x=8}
B→9 {B.x=9}

مثال۳-۸۲ به درخت تجزیه و نحوه ترجمه رشته 6-9+3 که در شکل ذیل ارائه شده است دقت کنید.



شکل ۳-٦۸ درخت تجزیه همراه صفات

پیاده سازی ترجمه

ترجمه ضمن تجزیه انجام می‌شود. به همین جهت ابتدا باید برای یک گرامر تجزیه کننده را ساخت و سپس دستورات ترجمه مناسب را در تجزیه کننده قرار داد.

مثال ۳-۸۳ برای گرامر ذیل تجزیه کننده پیشگو بسازید.

A→ B T
T → + B T | - B T | ∈
B→1
B→2
B→3
B→4
B→5
B→6
B→7
B→8
B→9

تجزیه کننده به صورت ذیل است.

```
char lookahead;
void A(){
B();
T();
}

T(){
if(lookahead=='+'){
lookahead=getche();
B();
T();
}
else  if(lookahead=='-'){
lookahead=getche();
B();
T();
}
else ;
return;
}

B(){
if(lookahead>'0' && lookahead<='9')
   lookahead=getche();
}
main(){
lookahed=getche();
A();
return;
}
```

با توجه به قوانین ترجمه ذیل دستورات ترجمه را در تجزیه کننده قرار می‌دهیم.

```
A→ B T
T → + B { printf("+");} T | - B { printf("+");}T | ∈
B→1  { printf("1");}
B→2  { printf("2");}
B→3  { printf("3");}
B→4  { printf("4");}
B→5  { printf("5");}
B→6  { printf("6");}
B→7  { printf("7");}
B→8  { printf("8");}
B→9  { printf("9");}
```

قوانین ترجمه به صورت ذیل به گرامر اضافه می‌شوند.

```
char lookahead;
void A(){
B();
T();
}

T(){
if(lookahead=='+'){
```

```
lookahead=getche();
B();
printf("+");
T();
}
else if(lookahead=='-'){
lookahead=getche();
B();
printf("-");
T();
}
else ;
return;
}

B(){
if(lookahead>'0' && lookahead<='9'){
   printf("%c",lookahead);
   lookahead=getche();
}
}
main(){
lookahed=getche();
A();
return;
}
```

در تجزیه کننده‌های پایین به بالا از تولید کننده تجزیه کننده استفاده می‌گردد، که چگونگی و نحوه نوشتن قوانین ترجمه به نوع تولید کننده مورد استفاده بستگی دارد.

در فصل یک توضیح داده شد که بعد از تحلیل لغوی و نحوی، تحلیل معنایی انجام می‌شود، روشی که در ترجمه مورد استفاده قرار گرفت در تحلیل معنایی نیز قابل استفاده است. یعنی در ضمن انجام تجزیه می‌توان تحلیل معنایی و تست‌های لازم را انجام داد. این مطلب نشان می‌دهد که تحلیل لغوی، نحوی، معنایی و ترجمه به طور کامل از یکدیگر جدا نیستند بلکه با یکدیگر و با هم انجام می‌شوند.

۳-۲۱ ساخت یک تجزیه کننده

برای ساخت تحلیلگر نحوی روشهای مختلفی وجود دارد که برخی از آنها عبارتند از:

۱- ساخت تحلیلگر نحوی با استفاده از زبانهای برنامه سازی سطح بالا مانند زبانهایC و پاسکال .

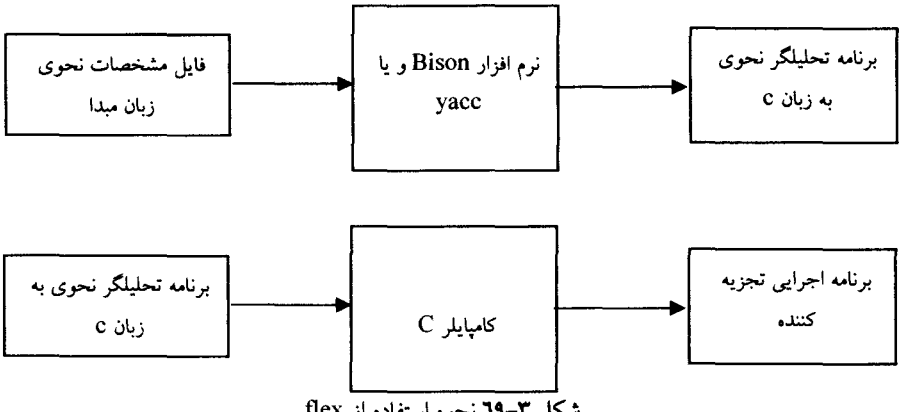
در این روش با استفاده از زبانهای سطح بالا تجزیه کننده را پیاده سازی می‌کنیم. به عنوان مثال در بخشهای گذشته تجزیه کننده پیشگو با زبان C پیاده سازی گردید. استفاده از این روش برای پیاده سازی گرامرهای پیچیده زمانبر است.

۲- استفاده از ابزارهای تولید کننده تجزیه کننده برای ساخت تحلیلگر نحوی

با توجه به اینکه ساخت تحلیلگر نحوی از اصول و روشهای یکسانی پیروی می‌کند در نتیجه می‌توان ابزاری ایجاد کرد که با اخذ مشخصات زبان مبدا تجزیه کننده آن را تولید کند. از جمله این ابزارها می‌توان به yacc[1] و Bison[2] اشاره کرد[3]. در Unix از yacc و در Windows از Bisonاستفاده می‌شود. نحوه استفاده از yacc و Bison به اندازه زیادی یکسان است. در نتیجه در ادامه به بررسی Bison می‌پردازیم. مفاهیم ارائه شده برای Yacc نیز قابل استفاده است.

۳-۲۲ تولید کننده تجزیه کننده

Bison برای گرامر، یک تجزیه کننده پایین به بالا به روش LALR(1) تولید می‌کند. برای تولید تجزیه کننده زبان مبدا، با استفاده از Bison ،کامپایلر نویس مشخصات گرامر زبان را با استفاده از گرامرهای مستقل از متن در یک فایل متنی با پسوند y. بیان می‌کند. سپس این فایل بوسیله نرم افزار Bison (یا yacc) به زبان C ترجمه می‌گردد، و در نهایت برنامه به زبان C با استفاده از کامپایلر C، برنامه اجرایی که، تجزیه کننده زبان مبدا است، تولید می‌گردد. شکل ذیل این مراحل را نشان می‌دهد.



شکل ۳-۶۹ نحوه استفاده از flex

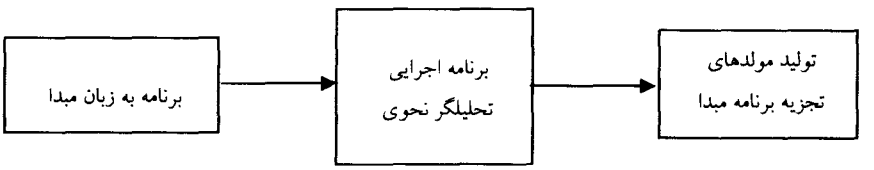
---

1. Yet Another Compiler Compiler

۲. Bison به معنی بوفالو یا گاومیش کوهان دار آمریکایی است.

۳. ابزارهای دیگری نیز تولید شده اند که از جمله می‌توان به LLgen اشاره کرد. LLgen تجزیه کننده بالا به پایین تولید می‌کند.

پس از این مرحله تجزیه کننده تولید شده است و از آن می‌توان به روش نشان داده شده در شکل ذیل استفاده کرد.



شکل ۳-۷۰ تولید برنامه اجرایی

با توجه به روند ذکر شده اولین قدم در تولید تجزیه کننده ایجاد فایل مشخصات نحوی زبان مبدا با استفاده از گرامر مستقل از متن است.

فایل مشخص کننده ساختار نحوی زبان مبدا به سه قسمت اصلی تقسیم می‌شود. هر قسمت از قسمت‌های دیگر با علامت %% جدا می‌گردد. سه قسمت اصلی به صورت ذیل بیان می‌شود.

```
اعلانها
%%
قوانین ترجمه
%%
توابع
```

۱- اعلانها: اعلانهای لازم برای اجرای برنامه بیان می‌گردد. این قسمت شامل دو بخش است که عبارتند از:

- اعلانهای مورد استفاده توابع: این قسمت بین }% و {% بیان می‌شود. به مثال ذیل دقت کنید.

```
%{
#include<string.h>
#include<math.h>
%}
```

- اعلان نشانه‌ها: این قسمت بعد ازاعلانها و بعد از {% و قبل از %% قرار می‌گیرد. در این قسمت نشانه‌های مورد استفاده در گرامر معرفی می‌شوند. به عنوان مثال به اعلان نشانه ذیل دقت کنید.

```
%{
#include<string.h>
#include<math.h>
```

```
%}
%token   NUM
%token   PLUS
%%
```

با این اعلان، تولید کننده تجزیه کننده Bison (yacc)، NUM و PLUS را به عنوان پایانه (نشانه در تحلیلگر لغوی) در نظر می‌گیرند.

۲- قوانین ترجمه: در این قسمت ساختار نحوی زبان مبدا به وسیله گرامرهای مستقل از متن بیان می‌شود. هر گرامر به صورت ذیل بیان می‌شود.

نام غیر پایانه : قاعده تولید اول غیر پایانه { قوانین ترجمه };

اگر انطباقی برای غیر پایانه یافت شود، قوانین ترجمه اجرا می‌گردد. پایان قواعد تولید مربوط به قاعده تولید به علامت ; ختم می‌گردد. اگر یک غیر پایانه بیش از یک قاعده تولید داشته باشد. می‌توان این قاعده تولید را با علامت | به صورت ذیل از یکدیگر جدا کرد.

```
مولد اول غیر پایانه : نام غیر پایانه  {قوانین ترجمه به زبان C مربوط به قاعده تولید اول }
                    | مولد دوم غیر پایانه  {قوانین ترجمه به زبانC مربوط به قاعده تولید دوم }
                    | مولد سوم غیر پایانه  {قوانین ترجمه به زبان C مربوط به قاعده تولید سوم }
                    .
                    .
                    .
                    ;
```

برای درک بهتر مطلب به مثال ذیل دقت کنید. قاعده تولید گرامر$T\rightarrow A+B|a$ در فایل مشخصات به صورت ذیل پیاده سازی می‌کنیم.

```
T:  A + B { printf("T")}
  | a     { printf("a")}
;
```

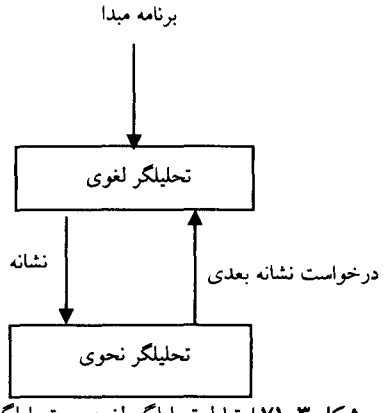
با توجه به قانون ترجمه فوق اگر انطباقی برای $T\rightarrow A+B$ یافت شود T و اگر انطباقی برای $T\rightarrow a$یافت شود a چاپ می‌گردد. برای نمایش گرامری به صورت $A\rightarrow B|\in$ از عبارت ذیل استفاده می‌کنیم.

```
A: B | ;
```

۳-توابع: در این قسمت روالهای مورد استفاده در ساخت تحلیلگر نحوی تعریف می‌شوند.

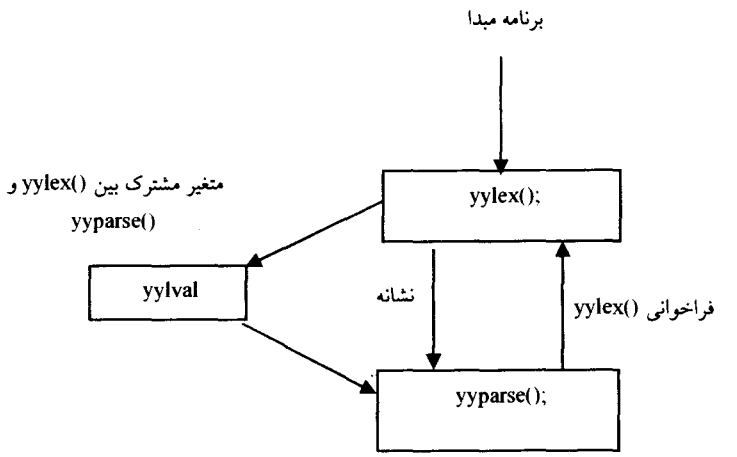
۳-۲۲-۱- ساختار تجزیه کننده تولید شده

ارتباط بین تحلیلگر لغوی و تحلیلگر نحوی در شکل ذیل نشان داده شده است



شکل ۳-۷۱ ارتباط تحلیلگر لغوی و تحلیلگر نحوی

تحلیلگر لغوی یک تابع است که توسط تجزیه کننده (تحلیلگر نحوی) فراخوانی می‌گردد. تحلیلگر نحوی هر گاه به پایانه احتیاج داشته باشد تحلیلگر لغوی را فراخوانی می‌کند.
Bison تحلیلگر نحوی را در تابعی به نام ;()yyparse پیاده‌سازی می‌کند. ()yyparse نیز تابع ()yylex را به عنوان تحلیلگر لغوی در نظر می‌گیرد و هرگاه ;()yyparse نیاز به پایانه (نشانه) داشته باشد ()yylex را فراخوانی می‌کند. پیاده‌سازی تابع ()yylex بر عهده برنامه‌نویس است. به همین دلیل مهمترین تابعی که در قسمت توابع باید پیاده‌سازی گردد تابع ;()yylex است.
هر گاه ()yyparse ، تابع ()yylex را فراخوانی می‌کند. ;()yylex نشانه را بوسیله دستور return به ()yyparse برمی‌گرداند. برای ارسال مقدار نشانه از متغیر مشترک yylval بین ()yylex و ()yyparse استفاده می‌گردد.



شکل ۳-۷۲ متغیر مشترک yylval

برنامه نویس تابع main() را نیز پیاده‌سازی می‌کند. مهمترین دستوری که در تابع main() باید قرار گیرد دستور فراخوانی yyparse() است.

```
int main(){
yyparse();
return 0;
}
```

برنامه نویس می‌تواند قبل یا بعد از فراخوانی yyparse(); دستورات لازم دیگری مانند مقداردهی اولیه به متغیرها را نیز اضافه کند.

yyparse() به گونه‌ای پیاده سازی می‌گردد که هر گاه با خطایی مواجه گردید، تابع yyerror(); را فراخوانی می‌کند. پیاده‌سازی این تابع نیز بر عهده برنامه‌نویس است. برنامه‌نویس باید yyerror() را در قسمت توابع پیاده‌سازی کند. اگر ایده خاصی برای پیاده‌سازی yyerror() ندارد می‌تواند از پیاده‌سازی ذیل استفاده کند.

```
yyerror()
{}
```

با توجه به مطالب ذکر شده برای گرامر ذیل با استفاده از Bison تجزیه کننده تولید می‌کنیم.

```
input → exp '\n'
exp →  exp + exp
        | exp - exp
        | exp * exp
        | exp / exp
        | ( exp )
        | 1
        | 2
        | 3
        | 4
```

```
| 5
| 6
| 7
| 8
| 9
```

این گرامر عبارات محاسباتی اعداد یک رقمی مانند 1+2 یا 1/2*4-9 را تولید می‌کند. با توجه به آنچه ذکر شد، ابتدا باید تحلیلگر لغوی با ;yylex() را ایجاد کنیم تا مورد استفاده، تجزیه کننده قرار گیرد. نشانه‌هایی که تحلیلگر لغوی باید تشخیص دهد پایانه‌های گرامر است. پایانه‌های گرامر عبارتند از:

- ارقام 1 الی9
- عملگرهای -,+,^,/,*
- علامت '\n' که نشان دهنده پایان رشته ورودی است.

;yylex() باید این پایانه‌ها را از ورودی تشخیص داده و به;yyparse() بر می‌گرداند. با توجه به آنچه ذکر شد ;yylex() را به صورت ذیل پیاده سازی می‌کنیم.

```
yylex(){
char ch;
ch=getc(stdin);
if(ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='^' || ch==')' || ch== '(' || '\n')
                    return(ch);
else if (ch>='1' && ch<='9')
                    return(ch);
}
```

تابع ;yyerror() را به صورت ذیل پیاده سازی می‌کنیم.

```
yyerror(){
printf("Error");
}
```

با استفاده از notepad ، فایلی به نام exp.y را با محتوای ذیل ایجاد می‌کنیم.

```
 %{
#include<stdio.h>
%}

%%
   input:  exp  '\n';
   exp:   | exp '+' exp
          | exp '-' exp
          | exp '*' exp
          | exp '/' exp
          | exp '^' exp
          | '(' exp ')'
|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
   ;
%%
main(){
yyparse();
}
```

```
yyerror(){printf("Error");}

yylex(){
char ch;
ch=getc(stdin);
if(ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='^' || ch==')' || ch== '(' || ch=='\n')
        return(ch);
else if(ch >= '1' && ch <= '9')
        return(ch);
}
```

با دستور ذیل این برنامه را با Bison کامپایل می‌کنیم.

```
c:\bison\bison   exp.y
```

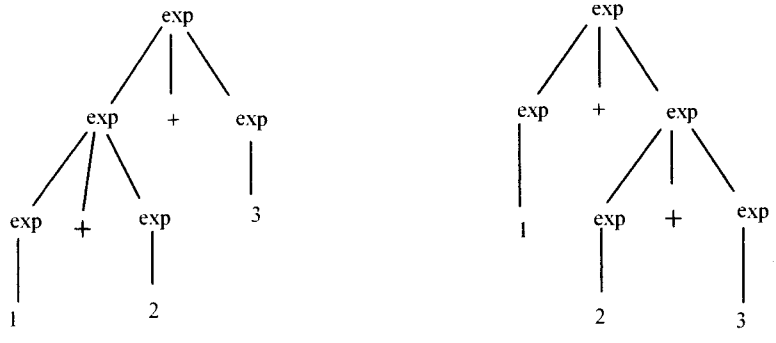
با اجرای این دستور پیغام ذیل ارائه می‌گردد.

```
contains 25 shift/reduce conflicts
```

این پیغام نشان می‌دهد که تولید کننده تجزیه کننده، با برخورد انتقال/کاهش مواجه شده است. همانطور که قبلا ذکر شد برای رفع برخورد چند روش وجود دارد، که عبارتند از:

- تغییر گرامر
- استفاده از قوانین جانبی مانند: استفاده از اولویت بندی و شرکت پذیری

برای گرامر مورد نظر از اولویت استفاده می‌کنیم. در بخش اعلانات عملگرها را به ترتیب اولویت پایین تر معرفی می‌کنیم. عملگری که اولویت کمتری دارد در ابتدا قرار می‌گیرد. اما علی رغم تعیین اولویت برای عملگرها بازهم مشکل حل نمی‌شود. به عنوان مثال برای رشته 1+2+3 می‌توان چند درخت تجزیه به صورت ذیل ایجاد کرد.



شکل ۳-۷۳ شرکت پذیری چپ و راست

همانطور که ملاحظه گردید علی رغم تعیین اولویت اگر عملگرها از یک نوع باشند، می‌توان چند درخت تجزیه ایجاد کرد. برای رفع ابهام از شرکت پذیری استفاده می‌کنیم. برای این منظور از عبارت %left برای تعیین شرکت پذیری چپ استفاده می‌شود. اغلب عملگرها شرکت پذیری چپ دارند، اما اگر عملگری شرکت پذیری راست داشته باشد مانند عملگر

توان می‌توان از عبارت %right برای تعیین شرکت پذیری راست استفاده کرد. پس از این تغییرات برنامه نهایی به صورت ذیل خواهد شد. فایل exp.y را به صورت ذیل تغییر می‌دهیم.

```
%{
#include<stdio.h>
%}
%left '+' '-'
%left '*' '/'
%right '^'
%%
    input:  exp  '\n';
    exp:    | exp '+' exp
            | exp '-' exp
            | exp '*' exp
            | exp '/' exp
            | exp '^' exp
            | '(' exp ')'
|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
    ;
%%
main(){
yyparse();
}
yyerror(){printf("Error");}

yylex(){
char ch;
ch=getc(stdin);
if(ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='^' || ch==')' || ch== '('  || ch=='\n')
            return(ch);
else if(ch >= '1' && ch <= '9')
            return(ch);
}
```

به منظور تشریح عملکرد تولید کننده تجزیه کننده یک مثال کامل را بررسی می‌کنیم. در این مثال می‌خواهیم تحلیلگری ایجاد کنیم تا عبارات محاسباتی شامل ^,-,+,/ و * را دریافت کرده و محاسبات را انجام داده و نتیجه را چاپ کند. در این مثال اعداد چند رقمی هستند. برای این منظور ابتدا باید ساختار این عبارات را به وسیله گرامرهای مستقل از متن نشان می‌دهیم. گرامر این زبان به صورت ذیل است.

```
input → exp
exp → exp + exp
          | exp - exp
          | exp * exp
          | exp / exp
          | NUM
```

NUM نشانه ای است که هنگام تشخیص یک عدد چند رقمی از تحلیلگر لغوی به تحلیلگر نحوی یا تجزیه کننده ارسال می‌گردد در نتیجه تحلیلگر لغوی یا ;yylex() را به صورت ذیل ایجاد می‌کنیم.

```
yylex(){
char ch;
int sum=0;
ch=getc(stdin);
while(ch==' ') c=getc(stdin);
if(ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='^' || ch==')' || ch== '(')
                    return(ch);
while(ch>='0' && ch<='9'){
   sum=sum*10 + (ch-'0');
   ch=getc(stdin);
}
ungetc (ch,stdin);
yylval=sum;
return(NUM);
}
```

;yylex() هرگاه یک عدد را تشخیص می‌دهد NUM را به تجزیه کننده برمی‌گرداند، مقدار عدد نیز به وسیله متغیر مشترک yylval به تحلیلگر نحوی ارسال می‌شود. اگر یک عملگر تشخیص داده شود کد اسکی عملگر به عنوان نشانه به تجزیه کننده برمی‌گردد. برای اینکه NUM نیز از نظر تجزیه کننده یک نشانه در نظر گرفته شود از عبارت ذیل در فایل exp.y استفاده می‌کنیم.

```
%token  NUM
```

برای گرامر با استفاده از ویرایشگر متن (مانند notepad) فایلی به نام exp.y ایجاد کرده و ساختار فوق را به صورت ذیل درون آن تعریف می‌کنیم.

```
%{
 #include <stdio>
%}

%token  NUM
%left  '+' '-'
%left  '*'  '/'
%right  '^'

%%
   input:   exp { printf("\t%d\n", $1); }
   ;
   exp:    NUM
         | exp '+' exp
         | exp '-' exp
         | exp '*' exp
         | exp '/' exp
         | exp '^' exp
         | '(' exp ')'
```

```
   %%
main(){
yyparse();
}
yyerror(){}
yylex(){
char ch;
int sum=0;
ch=getc(stdin);
while(ch==' ') c=getc(stdin);
if(ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='^' || ch==')' || ch== '(')
                return(ch);
while(ch>='0' && ch<='9'){
  sum=sum*10 + (ch-'0');
  ch=getc(stdin);
}
ungetc (ch,stdin);
yylval=sum;
return(NUM);
}
```

۳-۲۲-۲- ترجمه

برای ترجمه ورودی به خروجی از قوانین ترجمه در گرامر استفاده می‌کنیم. درون قوانین ترجمه برای هر غیرپایانه و پایانه در قاعده تولید متغیری وجود دارد که حاوی لغت منطبق شده بر آن غیرپایانه و یا پایانه است. این متغیر قابل دسترسی است و در صورت نیاز می‌توان از مقدار آنها استفاده کرد و یا تغییر داد. این متغیرها صفات غیر پایانه‌ها هستند که در بخشهای قبلی معرفی شد.

متغیر $$ حاوی لغت منطبق شده بر غیرپایانه سمت چپ قاعده تولید است.$1 حاوی لغت منطبق شده بر اولین پایانه یا غیر پایانه سمت راست قاعده تولید و $2 حاوی لغت منطبق شده بر دومین پایانه یا غیر پایانه سمت راست مولد و به همین ترتیب $i حاوی لغات لغت منطبق شده بر iامین پایانه یا غیرپایانه سمت راست مولد است. به مثال ذیل توجه کنید.

exp : exp + term

$$ صفت exp سمت چپ و $1 صفت exp سمت راست، $2 صفت حاوی + و $3 صفت term می‌باشد. از این صفات می‌توان برای تولید رشته خروجی استفاده کرد.

مثال۳-۸٤ به ترجمه دقت کنید.

```
exp : exp + term {$$=$1+ $3}
```

قانون $$=$1+$3 باعث می‌شود در صورت کاهش exp→exp+term مقدار صفت exp سمت راست با صفت term جمع شده و نتیجه در صفت exp سمت چپ قاعده تولید ذخیره می‌گردد. برای هر قاعده تولید گرامر عبارات محاسباتی قانون ترجمه را به صورت ذیل ایجاد می‌کنیم.

```
exp:   NUM          { $$ = $1; }
```

```
        | exp '+' exp       { $$ = $1 + $3; }
        | exp '-' exp       { $$ = $1 - $3; }
        | exp '*' exp       { $$ = $1 * $3; }
        | exp '/' exp       { $$ = $1 / $3; }
        | exp '^' exp       { $$ = pow ($1, $3); }
        | '(' exp ')'       { $$ = $2}
```

متن کامل برنامه به صورت ذیل است.

```
%{
 #include <math.h>
 #include <conio.h>
%}

%token  NUM
%left  '+' '-'
%left  '*'  '/'
%right  '^'

   %%
   input:  exp { printf("\t%d\n", $1); }   ;

   exp:   NUM       { $$ = $1; }
        | exp '+' exp       { $$ = $1 + $3; }
        | exp '-' exp       { $$ = $1 - $3; }
        | exp '*' exp       { $$ = $1 * $3; }
        | exp '/' exp       { $$ = $1 / $3; }
        | exp '^' exp       { $$ = pow ($1, $3); }
        | '(' exp ')'       { $$ = $2}
   ;
   %%
main(){
yyparse();
}
yyerror(){}
yylex(){

char ch;
int  sum=0;
ch=getc(stdin);
while(ch==' ') ch=getc(stdin);
if(ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='^' || ch==')' || ch== '('))
                  return(ch);
while(ch>='0' && ch<='9'){
   sum=sum*10 + (ch-'0');
   ch=getc(stdin);
}
ungetc (ch,stdin);
yylval=sum;
return(NUM);
}
```

پس از ترجمه این برنامه به زبان C ، فایل تولید شده را توسط کامپایلر C به فایل اجرایی تبدیل می‌کنیم. فایل اجرایی تولیدی حاوی تجزیه کننده و تحلیلگر لغوی می‌باشد. اگر رشته ورودی 4*3+2 باشد خروجی 14 است. در واقع رشته ورودی به 14 ترجمه شده است.

مثال۳-۸۵ اگر ترتیب نشانه‌ها به صورت ذیل تعریف شود و ورودی 4*3+2 باشد، خروجی چه عددی خواهد بود.

```
%token  NUM
%left  '*'  '/'
%left  '+' '-'
%right  '^'
```

با توجه به این ترتیب تقدم و اولویت + از * بیشتر است و در نتیجه عبارت 4*3+2 به صورت 4*(2+3) محاسبه خواهد شد.

۳-۲۲-۳- پیاده سازی yylex با flex

در مثالهای قبلی تحلیلگر لغوی به وسیله برنامه نویس و توسط ()yylex پیاده سازی شد. اگر زبان پیچیده باشد،تولید تحلیلگر لغوی به روش ذکر شده زمانبر است. lex/yacc و flex/Bison نرم افزارهایی هستند که برای تولید کامپایلر استفاده می‌گردند. نرم افزارهای lex و flex برای تولید تحلیلگر لغوی و نرم افزارهای yacc و Bison برای ساخت تجزیه کننده استفاده می‌شوند. این نرم افزارها با یکدیگر هماهنگ هستند. در نتیجه می‌توان با استفاده از lex و یا flex تحلیلگر لغوی(()yylex) را تولید کرد تا مورد استفاده Bison و yacc قرار گیرد. به عنوان مثال گرامر ذیل را در نظر می‌گیریم.

```
line→   '\n'  | exp '\n'
exp→    NUM
          | exp + exp
          | exp - exp
          | exp * exp
          | exp / exp
          | exp ^ exp
          | ( exp )
```

ابتدا پایانه‌های این گرامر را تعیین می‌کنیم. پایانه‌ها عبارتند از:

```
NUM   +  -  *  /  ^  (  )  \n
```

این پایانه‌ها، نشانه‌هایی هستند که از طرف تحلیلگر لغوی باید تشخیص داده شده و به تجزیه کننده بازگردانده شود. به همین جهت فایلی به نام exp.l ، به صورت ذیل ایجاد می‌کنیم. که این نشانه‌ها را از رشته ورودی تشخیص دهد.

```
%option noyywrap
%%
"("             return('(');
")"             return(')');
"+"             return('+');
"-"             return('-');
"*"             return('*');
"/"             return('/');
"^"             return('^');
"\n"            return('\n');
```

```
[0-9]+          {yylval=atoi(yytext);return(NUM);}
%%
```

فایل exp.l را با استفاده از flex بوسیله دستور ذیل کامپایل می‌کنیم.

```
flex exp.l
```

در نتیجه اجرای این دستور فایل lexyy.c تولید می‌گردد. lexyy.c برنامه تحلیلگر لغوی است.

با توجه به گرامر، فایل exp.y را به صورت ذیل ایجاد می‌کنیم.

```
%{
#include <math.h>
#include <stdio.h>
%}
%token NUM
   %left '-' '+'
   %left '*' '/'
   %right '^'
   %%
   input:   /* empty string */
            | input line
   ;

   line:    '\n'
            | exp '\n'  { printf("%d\n", $1); } ;

   exp:     NUM             { $$ = $1;         }
            | exp '+' exp     { $$ = $1 + $3;    }
            | exp '-' exp     { $$ = $1 - $3;    }
            | exp '*' exp     { $$ = $1 * $3;    }
            | exp '/' exp     { $$ = $1 / $3;    }
            | exp '^' exp     { $$ = pow ($1, $3); }
            | '(' exp ')'     { $$ = $2;         }    ;
   %%
#include <lexyy.c>
yyerror(){printf("error in input");}
int main()
{
 yyparse();
 return 0;
}
```

در exp.y تابع yylex(); تعریف نمی‌شود بلکه با استفاده از دستور ذیل تحلیلگر لغوی تولید شده توسط flex استفاده می‌گردد.

```
#include <lexyy.c>
```

فایل exp.y را با استفاده از دستور ذیل بوسیله Bison به زبان C ترجمه می‌کنیم.

```
bison exp.y
```

اجرای این دستور فایل exp.c را تولید می‌کند. با کامپایل exp.c فایل اجرایی exp.exe به دست می‌آید که هم شامل تحلیلگر لغوی و هم تجزیه کننده گرامر ماشین حساب است.

تمرینات

۱. بازگشتی چپ گرامر ذیل را حذف کنید.

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

۲. باتوجه به گرامر ذیل Follow را برای تمام غیر پایانه‌ها و first را برای سمت راست همه قواعد تولید محاسبه نمایید؟

$A \rightarrow BC$
$C \rightarrow aBC \mid \in$
$B \rightarrow DE$
$E \rightarrow bDE \mid \in$
$D \rightarrow cAd \mid e$

۳. برای گرامر ذیل تجزیه کننده پیشگو و پیشگوی غیر بازگشتی بسازید.

$E \rightarrow TB$
$B \rightarrow +TB \mid \in$
$T \rightarrow FC$
$C \rightarrow *FC \mid \in$
$F \rightarrow (E) \mid id$

٤. جدول تجزیه پیشگوی غیر بازگشتی گرامر ذیل را بسازید.

$S \rightarrow Abc$
$A \rightarrow aA \mid c$
$B \rightarrow b$
$C \rightarrow c$

۵. آیا گرامر ذیل LL(1) است؟

$A \rightarrow BCD$
$B \rightarrow bB \mid D$
$C \rightarrow c$
$D \rightarrow d$

٦. آیا گرامر ذیل LL(1) است؟

$A \rightarrow BCD$
$B \rightarrow bB \mid D$
$C \rightarrow Bc$
$D \rightarrow d$

۷. آیا گرامر ذیل LR(0) است در صورت مثبت بودن جواب، جدول تجزیه LR(0) برای گرامر ذیل بسازید.

$E \rightarrow E+T \mid T$
$T \rightarrow i \mid (E)$

۸ آیا گرامر ذیل SLR(1) است در صورت مثبت بودن جواب، جدول تجزیه SLR(1) برای گرامر ذیل بسازید.

E→E+T | T
T→ i |(E)

۹. جدول تجزیه SLR(1) برای گرامر ذیل بسازید، و رشته‌های (id+id) و id*id(id+id) را با استفاده از جدول به دست آمده تجزیه کنید.

E→E+T | T
T→ T*F|F| (E)
F→id

۱۰. نشان دهید گرامر ذیل SLR(1) است.

E→E+T | T
T→ TF| F
F→E+ |a|b

۱۱. آیا گرامر ذیل LR(1) است. در صورت مثبت بودن جواب جدول تجزیه آن را رسم کنید.

S→(SS) |) | (

۱۲. نشان دهید گرامر ذیل SLR(1) نیست.

S→ AaB| B
A→ bB| d
B→ A

۱۳. آیا گرامر ذیل LR(1) است. در صورت مثبت بودن جواب جدول تجزیه LR(1) گرامر را بسازید.

S→ AA
A→ aA | b

۱٤. آیا گرامر ذیل LALR(1) است. در صورت مثبت بودن جواب جدول تجزیه LALR(1) گرامر را بسازید.

S→ AA
A→ aA | b

۱۵. آیا گرامر ذیل LR(1) است. در صورت مثبت بودن جواب جدول تجزیه LR(1) گرامر را بسازید.

B→ BbB | BaB | c

۱٦. آیا گرامر ذیل LR(1) است. در صورت مثبت بودن جواب جدول تجزیه LR(1) گرامر را بسازید.

A→ AA | b

۱۷. برنامه ای بنویسید که رشته ای را دریافت کند و مشخص کند آیا رشته ورودی توسط گرامر ذیل قابل تولید است یا خیر.

E→E+T | T
T→ TF| F

F→E+ |a|b

۱۸. برنامه ای بنویسید که رشته ای را از کاربر دریافت کند و مشخص کند آیا رشته توسط گرامر ذیل قابل تولید است یا خیر.

A→ AaB | B
B→BbD | D
D→dAe | f

۱۹. دستگیره را تعریف کنید. در تجزیه رشته abbcde با توجه به گرامر ذیل دستگیره‌ها را مشخص کنید.

A→aBCe
B→Bbc | b
C→d

۲۰. برای گرامر G جدول SLR به صورت ذیل ارائه شده است مراحل تجزیه رشته fbfaf را مرحله به مرحله نشان دهید و مشخص کنید در این تجزیه چند بار عمل شیفت انجام می‌شود.

حالت	action						goto		
	f	a	b	(	)	$	A	B	D
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

گرامر G به صورت ذیل است.

1)A→ AaB
2)A→ B
3)B→BbD
4)B→D
5)D→dAe
6)D→f

۲۱. در روش LR(1) با توجه به گرامر ذیل اگر I={[A→ .B,$]} باشد closure(I) را محاسبه کنید.

A→B
B→CC
C→bC|d

۲۲. برخورد انتقال/کاهش و کاهش/کاهش را در شرح داده و سپس نشان دهید گرامر ذیل به کدام نوع برخورد می‌رسد؟

A→abcA|abcAdA|e

۲۳. ماشین خودکار LR(0) را برای گرامر ذیل ارائه دهید؟

1 - E→E+T
2 - E→T
3 - T→T*F
4 - T→F
5 - F→ (E)
6 - F→id

۲٤. با توجه به قواعد ترجمه ذیل رشته 3*1/2*2 را ترجمه کنید.

مولد	قواعد معنایی
expr→expr$_1$*term	Expr.t = expr$_1$.t || term.t|| '*'
expr→expr$_1$/term	Expr.t = expr$_1$.t || term.t|| '/'
expr→term	Expr.t = term.t
term→0	Term.t = ' 0'
term→1	Term.t = ' 1'
term→2	Term.t = ' 2'
term→3	Term.t = ' 3'

۲۵. گرامر متناظر با تجزیه کننده ذیل را مشخص کنید.

```
#include <stdio.h>
#include <stdlib.h>

char lookahead;  // Source token

void getlookahead(void)
{ lookahead = getchar(); }

void accept(char expectedterminal, char *errormessage)
{ if (lookahead != expectedterminal)
  {
  puts(errormessage);
  exit(1);
  }
  getlookahead();
}

void B(void){
 switch (lookahead)
  { case 'a':
          getlookahead();
          break;
    case '(':getlookahead();
                        B();
```

```
                        while (lookahead == '+') {
                        getlookahead();
          B();
                          }
                    accept(')', " Error - ')' expected");
               break;
   case '[':getlookahead();
                        B();
                        accept(']', " Error - ']' expected");
                        break;
   case ')':
   case ']':
   case '+':
   case '.':
     break;   // no action for followers of B
   default:
     printf("Unknown symbol\n"); exit(1);
  }
}

void A(void)
{ B(); accept('.', " Error - '.' expected"); }

void main(void)
{ lookahead = getchar(); A();
  printf("Successful\n");
}
```

۲٦. گرامر متناظر با تجزیه کننده ذیل را مشخص کنید.

```
#include <stdio.h>
#include <stdlib.h>

char lookahead;
void getlookahead(void)
{ lookahead = getchar(); }

void accept(char expectedterminal, char *errormessage)
{ if (lookahead != expectedterminal) {
  puts(errormessage);
  exit(1); }
  getlookahead();
}

void A(void);
void B(void);
void C(void);
void D(void);
void A(void)
{ B(); accept('.', " Error - '.' expected"); }

void B(void)
  { switch (lookahead)
  { case 'a':getlookahead();
```

```
            break;
    case '(':getlookahead();
                    C();
                accept(')', " Error - ')'expected");
                break;
    case '[':getlookahead();
                                B();
                               accept(']', " Error - ']' expected");
                               break;
    case ')':
    case ']':
    case '+':
    case '.':break;
    default:printf("Unknown symbol\n"); exit(1);
   }
 }
 void C(void){ B(); D(); }
 void D(void){ while (lookahead == '+') {
                                          getlookahead();
                                          B();
                                          }
}
 void main()
 { lookahead =
getchar();
   A();
   printf("Successful\n");
 }
```

تستهای تکمیلی

۱- بدون استفاده از ترکیب جلوبندی و عقب بندی برای تولید کامپایلر، برای ۱۰ زبان که روی ۱۲ماشین مختلف باید اجرا شوند به چند کامپایلر نیاز است؟

الف- ۲۲ ب- ۱۲۰

ج- ٦۰ د- ۱۱

۲- با استفاده از جلوبندی و عقب بندی، برای تولید کامپایلر، برای ۱۰ زبان که روی ۱۲ماشین مختلف باید اجرا شوند در مجموع به چند جلوبندی و عقب بندی نیاز است؟

الف- ۲۲ ب- ۱۲۰

ج- ٦۰ د- ۲۰

۳- کدام یک از موارد ذیل جزء جلوبندی کامپایلر نیست؟

الف- تولید کننده کد نهایی ب- تحلیلگر معنایی

ج- تحلیلگر لغوی د- هیچکدام

٤- می دانیم که عمل جمع دو شناسه که یکی نام آرایه و دیگری نام یک روال باشد، خطا است. اگر در برنامه‌ای چنین موردی وجود داشته باشد، کدام یک از موارد ذیل این خطا را کشف خواهد کرد؟

الف- تحلیلگر لغوی ب- تحلیلگر نحوی

ج- تحلیلگر معنایی د- تولید کننده کد میانی

۵- کشف خطای مربوط به استفاده از شناسه‌ای که تعریف نشده است، از وظائف کدام یک از موارد ذیل است؟

الف- تحلیلگر لغوی ب- تحلیلگر معنایی

ج- تحلیلگر نحوی　　د- تولید کننده کد نهایی

٦- اگر کاربری در برنامه به زبان پاسکال بـه جـای عبـارت PROGRAM بـه اشـتباه عبـارت PRORAM (به جای لغت PROGRAM لغت PRORAM) را قرار دهد کدام یک از موارد ذیل این خطا را کشف می‌کند؟

الف- تحلیلگر لغوی　　ب- تحلیلگر نحوی

ج- تحلیلگر معنایی　　د- تولید کننده کد میانی

۷- عبارت ذیل به زبان پاسکال نوشته شده است. تحلیلگر لغوی چند نشانه برای عبارت ذیل تولید خواهد کرد؟

```
TEMP:='SCHOOL';
```

الف-۳　　ب- ٤

ج-٥　　د-٦

۸- کدام یک از موارد ذیل غلط است؟

الف- پردازش درشت دستورالعملها از وظائف پیش پردازشگر است.

ب- ضمیمه کردن فایلها به فایل اصلی برنامه از وظائف پیش پردازشگر است.

ج- خروجی پیش پردازشگر ورودی کامپایلر است.

د- تصحیح آدرسهای برنامه به منظور اجرا، از وظائف پیش پردازشگر است.

۹- در زبان پاسکال در عبارت A:='ali'; لغت 'ali' چه نوع لغتی است.

الف- شناسه　　ب- ثوابت

ج- علائم　　د- هیچکدام

۱۰-در زبان flex کدامیک از موارد ذیل رشته xy را می پذیرد؟

الف- [^ab]　　ب- xy{2,4}

ج- x/1　　د- yx

۱۱-کدامیک از مجموعه های ذیل را با استفاده از عبارات با قاعده می توان نشان داد؟

الف- مجموعه تمام رشته های پرانتزدار موازنه ای (رشته هایی که تعـداد پرانتزهـای بـاز بـا تعداد پرانتزهای بسته برابر باشد)

ب- {w رشته ای از 0 و 1 | w0w}

ج- مجموعه رشته های Hollerith که به صورت $nHa_1a_2a_3...a_n$ است.(در این نوع رشته، تعداد کاراکترهای a بعد از H برابر عدد n قبل از H است.)

د- هیچکدام

۱۲- کدامیک از موارد ذیل صحیح نیست؟

الف- عبارت با قاعده را می توان توسط گرامر مستقل از متن نیز بیان کرد.

ب- یکی از وظائف پیش پردازشگر ، پردازش درشت دستورالعملها است.

ج- کم بودن تعداد گذرها از ویژگیهای مطلوب کامپایلر ها است.

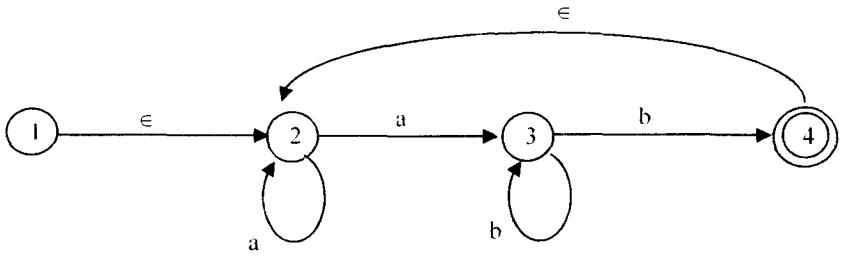
د- هیچکدام

۱۳- در زبان flex کدام یک از عبارات با قاعده ذیل رشته a را می پذیرد؟

الف- [^xyz] ب- x{1,3}

ج- xyz/123 د- هیچکدام

۱٤- با توجه به NFAی ذیل ∈_closure(4) کدام یک از موارد ذیل است(حالت ۱، حالت شروع است)؟



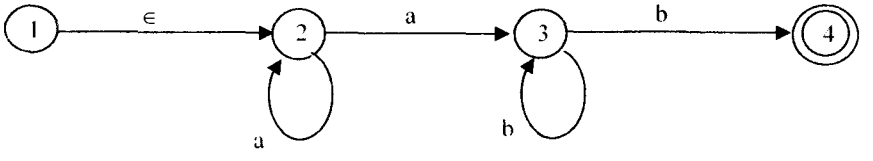
الف- {2}

ب- {3.2.4}

ج- {2.4}

د- {2.3.4}

۱۵- کدام یک از موارد ذیل هم ارز NFAی رسم شده است(حالت ۱، حالت شروع است)؟



الف- a+b+

ب- a*b*

ج- (ab)* د- (ab)(ab)*

۱۶- اگر n یک cat_node با سمت چپ c1 و سمت راست c2 باشد. آنگاه nullable(n) از کدام یک از روابط ذیل محاسبه می شود.

الف- nullable(c1) AND nullable(c2)

ب- nullable(c1) OR nullable(c2)

ج- nullable(c2)

د- nullable(c1)

۱۷- اگر n یک or_node با سمت چپ c1 و سمت راست c2 باشد. آنگاه Nullable(n) کدامیک از موارد ذیل است؟

الف- Nullable(c1)

ب- Nullable(c2)

ج- Nullable(c1) and Nullable(c2)

د- Nullable(c1) or Nullable(c2)

۱۸- اگر n یک cat_node با سمت چپ c1 و سمت راست c2 باشد و lastpos(c1)={2,8,3} و firstpos(c2)={5,6,7} باشد آنگاه مکانهای موجود درکدامیک از مجموعه های ذیل قطعا در followpos(2) قرار دارد؟

الف- {5,2,8} ب- {5,6,7}

ج- {2,8,3} د- {2,8,3,5,6,7}

۱۹- اگر n یک or_node با سمت چپ c1 و سمت راست c2 باشد. و lastpos(c1)={2,8,3} و firstpos(c2)={5,6,7} باشد آنگاه مکانهای موجود درکدامیک از مجموعه های ذیل قطعا در followpos(2) قرار دارد؟

الف- {5,2,8} ب- {5,6,7}

ج- {2,8,3} د- هیچکدام

۲۰- اگر n یک star_node باشد و lastpos(n)={2,4,6} و firstpos(n)={1,2,7} باشد آنگاه کدامیک از مجموعه های ذیل قطعا در followpos(4) است؟

الف- {1,2,7} ب- {2,4,6}

ج- {2}　　د- {1,2,7,4,6}

۲۱- کدام یک از موارد ذیل صحیح نیست؟

الف- جدول نماد برای نگهداری اطلاعاتی درباره ساختمانهای مختلف زبان مبدا به کار می رود.

ب- اطلاعات درون جدول نماد توسط فازهای تحلیل گردآوری می شود.

ج- اطلاعات درون جدول نماد توسط فازهای تولید کد برای تولید کد نهایی استفاده می شود.

د- هیچکدام

۲۲- اگر زبان L={a,aa,aaa} و زبان M={0,1} باشد، کدامیک از رشته های ذیل در زبان LUM است؟

الف- a　　ب- 01

ج- aaaa　　د- 1a

۲۳- اگر زبان L={a,aa,aaa} و M={0,1} باشد، کدامیک از رشته های ذیل در زبان L.M است؟

الف- 11　　ب- 01

ج- aaaa　　د- 0a

۲٤- برای کدامیک از عبارات با قاعده ذیل نمی توان یک DFA رسم کرد؟

الف- a*b*c*　　ب- $a^*b^+cb^*$

ج- abc*　　د- هیچکدام

۲۵- در زبان flex کدام یک از رشته‌های ذیل توسط عبارات b{2,4} پذیرفته نمی‌شود؟

الف- b　　ب- bb

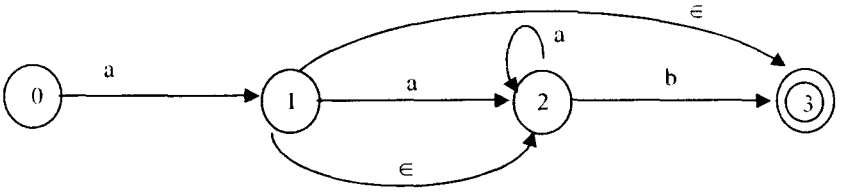
ج- bbb　　د- bbbb

۲٦- اگر $firstpos(c_2)$= {4} و $firstpos(c_1)$={1,2} و nullable(c1)=false و n یک cat-node با فرزند سمت چپ c1 و فرزند سمت راست c2 باشد آنگاه firstpos(n) کدام یک از موارد ذیل است؟

الف- {}　　ب- {4}

ج- {1,2}　　د- {1,2,4}

۲۷- اگر T={1,2} باشد با توجه به NFA ذیل ∈closure(mov(T,a))کدام یک ازموارد ذیل است (حالت ۱، حالت شروع است)؟



الف-{}

ب- {1,2,3}

ج- {0,1,2,3}

د- {2}

۲۸- کدامیک از مجموعه های ذیل را توسط عبارات باقاعده می‌توان نشان داد؟

الف-{w رشته ای از {WcW|b,a}

ب- مجموعه رشته‌هایی از b و a که تعداد b و a برابر باشند.

ج- مجموعه رشته‌هایی از a,b,c,d که در آنها هیچ حرفی تکرار نشده باشد.

د- {w رشته‌ای از $\{WW^R|b,a\}$

توجه : W^R یعنی معکوس W به عنوان مثال abc → cba

۲۹- کدام یک از موارد غلط است؟

الف- جدول نماد ساختاری است که برای نگهداری اطلاعات ساختمانهای مختلف زبان مبدا به کار می‌رود.

ب- تحلیلگر لغوی شناسه‌ها را تشخیص داده و در صورتیکه قبلا در جدول نماد وارد نشده باشد، برای آن وارده‌ای در جدول نماد ایجاد می‌کند

ج- کلمات کلیدی را می توان به صورت مقادیر اولیه وارد جدول نماد کرد.

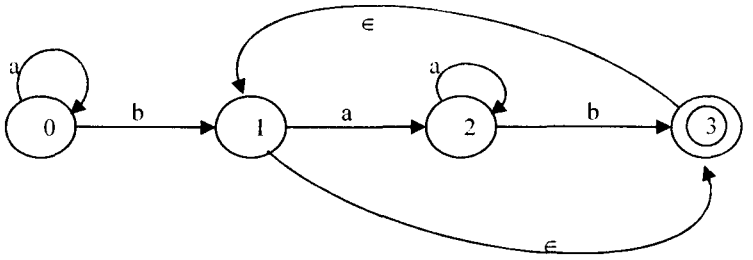
د- هیچکدام.

۳۰-عبارت باقاعده *(a|b) هم ارز کدام یک از موارد ذیل نیست؟

الف- $(aa^*|b)^*$ ب- $(a^*b^*)^*$

ج- $(a^*b)^*$ د- $((a|b)^*)^*$

۳۱- NFA ذیل هم ارز کدام یک از عبارات با قاعده ذیل است(حالت ۱، حالت شروع است)؟



الف- $a^*b(a^+b)^*$ ب- $a^+b(a^*b)^+$

ج- $a^+bb(a^*b)^+$ د- $a^*b(a^*b)b^*$

۳۲- اگر n یک cat_node با سمت چپ c1 و سمت راست c2 باشد و Nullable(c1)=True باشد، firstpos(n) کدام یک از موارد ذیل است؟

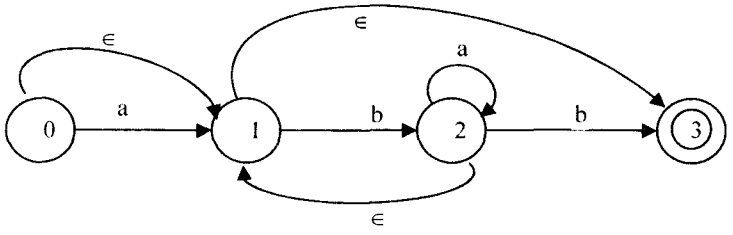
الف- Firstpos(c1) U Firstpos(c2)

ب- Firstpos(c1)

ج- Firstpos(c2)

د- Firstpos(c1) ∩ Firstpos(c2)

۳۳-با توجه به NFA ذیل ϵ-closure(2) کدام یک از موارد ذیل است؟



الف- {1,2,3} ب- {1,3}

ج- {0,1,2,3}　　د- {2,3}

۳٤-هم ارز گرامر ذیل کدام یک از موارد است؟

A→aA|bA|B
B→cD
D→aD|F
F→bF|∈

الف- (a|b) *ca*b*　　ب- $a^+b^+c(a|b)^*$

ج- $(a|b)^+ca^+b^+$　　د- $a^+b^+c(a|b)^+$

۳۵- برنامه ذیل تشخیص دهنده چه عبارت باقاعده ای است(به عبارت دیگر برای رشته های تولیدی کدام عبارت با قاعده پیغام accepted را چاپ می کند.

```
#inlcude<iostream.h>
#inlcude<stdlib.h>
#inlcude<stdio.h>
int main(){
int state;
char  ch;
state=1;
while(1){
switch(state){
        case 1: ch = getc(stdin);
                if ((ch>='a' && 'z'>=ch )|| (ch>='A' && 'Z'>=ch) )
                        state=2;
                else {
                        cout<<"Failed";
                        exit(0);
                    }
                break;
        case 2:ch = getc(stdin);
              if(ch=='\n'){
                        cout<<"Accepted";
                        exit(0);
                        }
      else if ((ch>='a' && 'z'>=ch) || (ch>='A' && 'Z'>=ch)||(ch>='0' && '9'>=ch ))
                state=2;
        else {
                cout<<"Failed";
                exit(0);
             }
        break;
  }
return(0);
}
```

الف- [a-zA-Z0-9]*　　ب- [a-zA-Z]*

ج- [a-zA-Z]+ د- [a-zA-Z][a-zA-Z0-9]*

۳۶- با توجه به قطعه برنامه flex ذیل اگر رشته ورودی abb باشد پیغام خروجی چیست.

```
%%
"ab"        {printf("a");}
"abb"       {printf("b");}
.           { printf("end");}
%%
```

الف- a ب- b

ج- end د- ab

۳۷- با توجه به قطعه برنامه flex ذیل اگر رشته ورودی 125 باشد پیغام خروجی چیست.

```
%%
"ab"        {printf("a");}
"abb"       {printf("b");}
.           { printf("end");}
%%
```

الف- a ب- b

ج- ۳ بار end چاپ می شود. د- پیغامی چاپ نمی شود.

۳۸- با توجه به قطعه برنامه flex ذیل اگر رشته ورودی begin باشد پیغام خروجی چیست.

```
%%
[a-zA-Z0-9]*  {printf("id");}
"begin"       { printf("begin");}
.             { printf("end");}
%%
```

الف- id ب- end

ج- begin د- هیچکدام

۳۹- در تولید جدول تجزیه به روش SLR(1) برای گرامر ذیل اگر I={E→.A} باشد، closure(I) شامل کدام یک از عناصر ذیل نیست؟

```
E→A
A→AaB | B
B→BbC | C
C → cAd | e
```

الف- C→ c.Ad ب- A→.B

ج- C→.e د- A→.AaB

۴۰- کدامیک از موارد ذیل صحیح نیست؟

الف- هیچ گرامر دارای بازگشتی چپ نمی تواند LL(1) باشد.

ب- هیچ گرامر LL(1) مبهم نیست.

ج- هر گرامر فاقد قاعده تولید ∈ که سمت راست هر قاعده تولید آن با یک پایانه مجزا شروع شود، همواره LL(1) است.

د- هیچکدام

۴۱- کدامیک از موارد ذیل صحیح نیست؟

الف- جداول SLR و LALR برای یک گرامر همواره دارای تعداد حالتهای یکسان می باشند.

ب- هر گرامر SLR(1) یک گرامر LR(1) نیز می باشد.

ج- هر گرامر SLR(1) غیر مبهم است.

د- هر گرامر غیر مبهمی SLR(1) است.

۴۲- کدامیک از موارد ذیل توسط گرامر مستقل از متن است قابل بیان است (راهنمایی: w رشته‌ای از a و b است)؟

الف- $\{wa^nb^nw\}$ ب- $\{www\}$

ج- $\{a^nb^nc^n\}$ د- هیچکدام

۴۳- کدام گزینه در مورد گرامر ذیل صحیح نیست؟

$S \rightarrow AaAb \mid BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

الف- این گرامر LL(1) است.

ب- این گرامر LR(0) است.

ج- این گرامر SLR(1) نیست.

د- هیچکدام

۴۴- با توجه به جدول تجزیه پیشگوی غیر بازگشتی ذیل در مورد تجزیه رشته bcbab کدام یک از موارد ذیل صحیح است؟

	a	b	c	(	)	$
A		A→CB		A→CB	B→∈	
B			B→cCB			B→∈
C		C→ED		C→ED		
D	D→aED		D→∈		D→∈	D→∈
E		E→b		E→(A)		

الف- این رشته پذیرفته می شود.

ب- این رشته پذیرفته نمی شود.

ج- اطلاعات داده شده برای تعیین پذیرفته شدن یا نشدن رشته کافی نیست.

د- با توجه به جدول گرامر مبهم است و در نتیجه نمی توان به این سوال پاسخ داد تا ابهام از گرامر رفع شود.

۴۵- پس از حذف بازگشتی‌های چپ از گرامر ذیل نتیجه کدام یک از موارد ذیل است؟

$A \rightarrow Bb \mid c$
$B \rightarrow Bd \mid Ae \mid \in$

الف-

$A \rightarrow Bb \mid c$
$B \rightarrow ceC \mid C$
$C \rightarrow dC \mid beC \mid \in$

ب-

$A \rightarrow Bb \mid c$
$B \rightarrow cC \mid Ce$
$C \rightarrow C \mid dbcC \mid \in$

د-

$A \rightarrow Bb \mid ce|C$
$B \rightarrow cC \mid C$
$C \rightarrow bcC \mid \in$

د-هیچکدام

۴۶- کدام یک از موارد ذیل صحیح نیست؟

الف- bison/yacc تولید کننده تجزیه کننده SLR است.

ب- در گرامر عملگر سمت راست هیچ قاعده تولیدی ∈ نمی باشد.

ج- هر گرامری را می توان به گرامر عملگر معادل آن تبدیل کرد.

د- هیچکدام

۴۷- کدام یک از ساختارهای زبان پاسکال را نمی توان توسط گرامر مستقل از متن بیان کرد؟

الف- ساختار if های تو در تو

ب- ساختارهای محاسباتی با پرانتزهای متوازن(یعنی تعداد پرانتز باز و بسته مساوی باشد)

ج- لزوم تعریف یک متغیر قبل از استفاده از آن

د- هیچکدام

۴۸- برنامه ذیل تجزیه کننده بازگشتی چه گرامری است(یا این برنامه به ازای رشته های تولیدی چه گرامری پیغام accepted را چاپ می کند)؟

(راهنمایی: getche() یک کاراکتر از ورودی می خواند. cout<<"accepted" و cout<<"error" به ترتیب پیغامaccepted و error چاپ می شود)

```
void B();
void  A( ){
char   ch;
ch=getche();
if(ch=='a') A( );
else  if(ch=='b')  B( );
     else if(ch=='c'){ cout<<"accepted";  return;}
          else {cout<<"error";return;}
}

void  B( ){
char   ch;
ch=getche();
if(ch=='b') B();
else if(ch=='c'){ cout<<"accepted";  return; }
else {cout<<"error";return; }
}
void main(){A(); return;}
```

الف-

$A \rightarrow aA \mid B$
$B \rightarrow bB \mid c$

ب-

$A \rightarrow aA \mid B \mid \epsilon$
$B \rightarrow bB \mid c$

ج-

$A \rightarrow aA \mid \in$
$B \rightarrow bB \mid c$

د- هیچکدام

۴۹- کدامیک از موارد ذیل غلط است ؟

الف- روش تجزیه پیشگو یک روش تجزیه پایین به بالا است.

ب- در روش تجزیه پیشگو مجموعه‌ای از رویه‌های بازگشتی برای پردازش رشته ورودی اجرا می گردند.

ج- در روش تجزیه پیشگو برای هر غیر پایانه از گرامر یک رویه نوشته می‌شود.

د- هیچکدام

۵۰- کدام گزینه در مورد گرامر ذیل صحیح است؟

$S \rightarrow Aa|bAc|dc|bda$
$A \rightarrow d$

الف- این گرامر LALR(1) است ولی SLR(1) نیست.

ب- این گرامر LALR(1) نیست و SLR(1) نیست.

ج- این گرامر LALR(1) است و SLR(1) است.

د- این گرامر LALR(1) نیست ولی SLR(1) است.

۵۱- اگر برای هریک از گرامرهای ذیل (بدون تغییر در گرامر) یک تجزیه کننده پیشگو تهیه شود، در تجزیه کننده کدامیک از گرامرهای ذیل ممکن است یک حلقه بی‌نهایت ایجاد شود؟

الف- $A \rightarrow aA|d$ ب- $S \rightarrow Sa|d$

ج- $B \rightarrow aB|bB|c|d$ د- $A \rightarrow aaA|bbA|ccA|d$

۵۲- کدام گزینه غلط است؟

الف-گرامرهای مبهمی وجود دارند که LL(1) هستند.

ب-گرامرهای دارای بازگشتی چپ LL(1) نمی باشند.

ج- گرامر مبهم، گرامری است که برای حداقل یک رشته تولیدی گرامر بتوان دو درخت تجزیه ایجاد کرد.

د- اگر در گرامری مولدی به صورت $A \rightarrow \alpha|\beta$ وجود داشته باشد به طوریکه β و α بتواند رشته تهی تولید کنند این گرامر LL(1) نیست.

۵۳- کدام گزینه غلط است ؟

الف- هر گرامر SLR(1) غیر مبهم است.

ب- هر گرامر غیر مبهی SLR(1) است.

ج- هر گرامر SLR(1) یک گرامر LR(1) نیز می‌باشد.

د- هیچکدام

٥٤- کدام گزینه غلط است ؟

الف- یک گرامر مبهم هرگز LR نیست.

ب- YACC یک تولید کننده تجزیه کننده LALR است.

ج- گرامرهای LR(1)ای وجود دارند که مبهم هستند.

د- اگر گرامری مبهم نباشد، هر شبه جمله راست از گرامر دقیقا یک دستگیره دارد.

۵۵- کدام گزینه غلط است ؟

الف- هر گرامر LL(1) یک گرامرLR(1) است.

ب- گرامرهای LR زبانهای کمتری نسبت به LL توصیف می‌کنند.

ج- گرامری که جدول تجزیه پیشگوی غیر بازگشتی آن دارای هیچ مدخلی با چند قاعده تولید نباشد، گرامر LL(1) است.

د- برای همه گرامرهای مستقل از متن نمی‌توان از تجزیهLR(0) استفاده نمود.

٥٦-کدام یک از موارد ذیل غلط است؟

الف- گرامرهای LL(1) مبهم نیستند.

ب- هر گرامری را می‌توان به LL(1) تبدیل کرد.

ج- اگر گرامری مبهم باشد جدول تجزیه پیشگوی غیر بازگشتی تولید شده از آن حداقل یک خانه با بیش از یک قاعده تولید دارد.

د- اگر با یک نماد از ورودی بتوان قاعده تولید بعدی را تشخیص داد برای این گرامر می‌توان تجزیه کننده پیشگو ساخت.

۵۷- follow(E) کدام یک از موارد ذیل است ؟

A→CB
B→aCB|∈
C→FE
E→bFE|∈
F→cAd|e

الف- {a,b,d, $}　　ب- {a,d,$ }

ج- {a,d}　　د- {a,c,d,$}

۵۸- کدام یک از گرامرهای ذیل،گرامر عملگر نیست ؟

الف- A→ABA|a
B→b

ب- S→Sa|a

ج- S→aSa|a

د- A→aA|B
B→bB|b

۵۹-با توجه به گرامر ذیل first(BC) شامل کدامیک از موارد ذیل نیست؟

A→BC | e
B→ bB | a | ∈
C→d

الف- b　　ب- e

ج- d　　د- a

۶۰- با توجه به قواعد ترجمه ذیل ترجمه رشته abbbc چیست؟

قوانین معنایی	مولد
A.x=B.x || C.x	A→ a BC
B.x=B.x || '1	B→ bB
B.x= '0	B→ b
C.x='3	C→c

الف- 1310　　ب- 0131

ج- 3110　　د- 0113

۶۱- کدامیک از موارد ذیل بازگشتی چپ مخفی دارد.

الف- A→Ac | d

ب- A→BA | d
B→AB | a

ج- A→BAc | d

B→d| ∈

د- A→B | d

B→c

٦٢-کدامیک از موارد ذیل بازگشتی چپ غیر مستقیم دارد.

الف- A→Ac | d

ب- A→BA | d

B→AB | a

۱- A→BAc | d

B→d| ∈

د- A→B | d

B→c

٦٣- کدامیک از گرامرهای ذیل مبهم نیست؟

الف- S→ietS|ietSeS|a

ب- S→bSS|aSS|a

ج- S→a|SS|Sa

د- S→0S1|01

٦٤- با توجه به الگوی ترجمه ذیل، اگر رشته ورودی dbdad باشد حاصل ترجمه کدامیک از موارد ذیل است ؟

```
A→AaB{print('1')}
A→AbB{print('2')}
A→B
B→d     {print('3')}
```

الف- 33231

ب- 33321

ج- 321

د- 231

٦٥- کدامیک از مجموعه های ذیل توسط گرامر مسقل از متن قابل توصیف نیست؟

الف- $\{a^n b^n c^m d^m | n>=1, m>=1\}$

ب- $\{a^n b^n | n>=1\}$

ج- $\{a^n b^m c^n d^m | m>=1, n>=1\}$

د- $\{wcw^R | w \text{ is in } (a|b)^*\}$ راهنمایی: w^R معکوس رشته w است.

٦٦-ایجاد حلقه بی نهایت در برنامه جزء کدام دسته از خطاهای ذیل است؟

الف- لغوی

ب- نحوی

ج- معنایی　　د- منطقی

۶۷- هم ارز گرامر ذیل کدامیک از موارد ذیل است؟

A→AC|BC
A→aA|a
B→bB|b
C→cC|∈

الف- (a*|b*)c*　　ب- (a|b)*c*

ج- (a⁺|b⁺)c*　　د- (a|b)*c⁺

۶۸- با توجه به گرامر ذیل در تجزیه پایین به بالا رشته abd کدامیک از موارد دستگیره نیست.

A→aA | eB | B
B→bA | d | cB

الف- aA　　ب- B

ج- bB　　د- eB

۶۹- با توجه به گرامر ذیل first(DE) شامل کدام یک از پایانه های ذیل نیست؟

A→DE|B
B→bB|∈
D→a|∈|e
E→c|∈

الف- b　　ب- a

ج- e　　د- c

۷۰- با توجه به گرامر ذیل follow(D) شامل کدام یک از پایانه های ذیل است؟

A→DE|B
B→bB|∈
D→a|∈
E→c|∈

الف- b　　ب- a

ج- e　　د- c

۷۱- محتوای مدخل M[E,c] جدول تجزیه پیشگوی غیر بازگشتی گرامر ذیل چیست؟

A→DE|B
B→bB|e
D→a|b
E→cB|∈

الف- A→DE　　ب- B→∈

ج- D→a|∈　　د- E→cB

۷۲- محتوای مدخل M[D,c] جدول تجزیه پیشگوی غیر بازگشتی گرامر ذیل چیست؟

A→DE|B
B→bB|∈
D→a|∈
E→c|∈

الف- A→DE　　ب- B→∈

ج- D→∈　　د- A→B

۷۳- با توجه به جدول تجزیه پیشگوی غیر بازگشتی ذیل کدامیک از موارد ذیل در مورد گرامر مورد نظر صحیح است؟

	a	b	C	$
A	A→aB		A→C	
B		B→b		
C			C→c	

الف- این گرامر LL(1) است.

ب- این گرامر LL(1) نیست.

ج- اطلاعات سوال برای تعیین LL(1) بودن کافی نیست.

د- هیچکدام

۷٤- کدامیک از گرامرهای ذیل LL(1) است.

الف- A→aA|aaA|b

ب- A→BC|∈
B→bB|∈
C→cC|∈

ج- S→Ab|a
A→bA|∈

د- S→abA|d
A→e

۷۵- کدامیک از گرامرهای ذیل برخورد first/first دارد.

الف- S→aS|bS|∈

ب- S→aaS|bbS|∈

ج- S→aS|aS|b

د- S→aaS|AS|∈

A→cb

۷۶- کدام یک از موارد ذیل در مورد گرامر ذیل صحیح است.

A→ bA | eB | B
B→ bB | d | cB

الف- این گرامر برخورد first/first دارد

ب- این گرامر برخورد first/follow دارد

ج- این گرامر بازگشتی چپ دارد.

د- هیچکدام

۷۷-کدامیک از موارد ذیل در مورد گرامر صحیح است.

A→ bA | eB | B
B→ bB | d | cB

الف- این گرامر LL(1) نیست.

ب- این گرامر SLR(1) است.

ج- این گرامر LR(1) است.

د- این گرامر LALR(1) است.

۷۸- کدامیک از گرامرهای ذیل برخورد first/follow دارد.

الف- S→bS|∈

ب- S→aS|bS|A

A→a

ج- S→Aa|ab

A→aA|∈

د- هیچکدام

۷۹- در تجزیه رشته 3*2+1 با استفاده از روش عملگر- اولویت با توجه به جدول اولویت ذیل چهارمین دستگیره کدام است.

E→E+E | E*E | a

	A	+	*	$
a		>	>	>
+		>	>	>
*			>	>
$				

الف- E*E　　ب- E+E

ج- E　　د- E+E*E

۸۰- جدول تجزیه SLR(1) گرامر ذیل چند سطر دارد.

A→aA | bA | c

الف- ۳　　ب- ٤

ج- ٥　　د- ٦

۸۱- با توجه به گرامر ذیل closure(S→.A) با استفاده از روش LR(0) کدام یک از مجموعه های ذیل است.

S→A
A→DE|B
B→bB|∈
D→a|∈
E→c|∈

الف- { S→.A , A→.DE , A→.B , B→.bB , B→. , D→.a , D→. }

ب- { A→.DE , A→.B , B→.bB , B→. , D→.a , D→. }

ج- { S→.A , A→.DE , A→.B , B→.bB , D→.a }

د- { S→A. , A→D.E , A→.B , B→.bB , B→. , D→.a , }

۸۲- با توجه به گرامر ذیل کدام یک از عناصر ذیل در closure(E→.E+E) با استفاده از روش SLR(1) قرار ندارد.

E→E+E | E-E | (E) | id

الف- E→.E+E　　ب- E→.E-E

ج- E→.id　　د- E→(E).

۸۳- با توجه به گرامر ذیل closure([S→.A,$]) به روش LR(1) کدام یک از مجموعه های ذیل است.

S→A
A→AA | a | ∈

الف- { [S→.A,$]}

ب- { [S→.A,$], [S→.AA,$a], [S→.a,$a], [S→.,a$]}

ج- { [S→.A,$], [S→.AA,$], [S→.a,$] }

د- { [S→.A.$], [S→.AA.$], [S→.a.$], [S→..$] }

۸۴- کدام یک از عناصر ذیل در closure([S→A.a,$]) به روش LR(1) است.

S→A
S→AB | a | Aa
B→A | bB | ∈

الف- [A→A.a.$]

ب- [A→AB.,a]

ج- [S→A.a,a]

د- [B→A..$]

۸۵- با توجه به گرامر ذیل کدامیک از موارد ذیل صحیح است.

A→aA | Aa | ∈

الف- این گرامر SLR(1) است ولی LR(0) نیست.

ب- این گرامر SLR(1) نیست ولی LR(0) است.

ج- این گرامر SLR(1) و LR(0) است.

د- این گرامر SLR(1) و LR(0) نیست.

۸۶- با توجه به گرامر ذیل کدامیک از موارد ذیل صحیح است.

A→aA | Aa | ∈

الف- این گرامر LR(1) است ولی LALR(1) نیست.

ب- این گرامر LR(1) نیست ولی LALR(1) است.

ج- این گرامر LR(1) و LALR(1) است.

د- این گرامر LR(1) و LALR(1) نیست.

۸۷- کدامیک از موارد ذیل صحیح نیست.

الف- هر گرامر SLR(1) یک گرامر LR(1) است.

ب- هر گرامر LR(1) یک گرامر LALR(1) است.

ج- هر گرامر LR(0) یک گرامر LALR(1) است.

د- هر گرامر LR(0) یک گرامر SLR(1) است.

۸۸- کدام یک از موارد ذیل در مورد گرامر ذیل صحیح است.

S→Aa | bAc | Bc | bBa
A→d
B→d

الف- گرامر LR(1) نیست ولی LALR(1) است.

ب- گرامر LR(1) است و LALR(1) است.

ج- گرامر LR(1) نیست و LALR(1) نیست.

د- گرامر LR(1) است ولی LALR(1) نیست.

۸۹- کدام یک از موارد ذیل در مورد گرامر ذیل صحیح است.

A→aAd | bBd | aBe | bAe
A→c
B→c

الف- گرامر LR(1) نیست ولی LALR(1) است.

ب- گرامر LR(1) است و LALR(1) است.

ج- گرامر LR(1) نیست و LALR(1) نیست.

د- گرامر LR(1) است ولی LALR(1) نیست.

۹۰- جدول تجزیه LR(1) گرامر ذیل چند سطر دارد.

A→aA | bA | c

الف-۳ ب- ٤

ج- ٥ د- ٦

۹۱- کدامیک از موارد ذیل در مورد گرامر ذیل صحیح است.

S→aACb
A→b | ∈
C→cC | ∈

الف- گرامر LL(1) نیست و SLR(1) نیست.

ب- گرامر LL(1) است اما SLR(1) نیست.

ج- گرامر LL(1) نیست اما SLR(1) است.

د- گرامر LL(1) است و SLR(1) است.

۹۲- تجزیه کننده های پایین به بالا درخت تجزیه را به چه ترتیبی می سازند.

الف- inorder ب- postorder

ج- preorder د- هیچکدام

۹۳- تجزیه کننده های بالا به پایین درخت تجزیه را به چه ترتیبی می سازند.

الف- inorder ب- postorder

ج- preorder　　د- هیچکدام

۹۴- جدول تجزیه LR(1) گرامر ذیل چند سطر دارد.

A→BB
B→bB | a

الف- ۸　　ب- ۹

ج- ۱۰　　د- ۱۱

۹۵- جدول تجزیه LALR(1) گرامر ذیل چند سطر دارد.

A→BB
B→bB | a

الف- ٤　　ب- ٥

ج- ٦　　د- ٧

۹٦- جدول تجزیه SLR(1) گرامر ذیل چند سطر دارد.

E→E+T
E→T
T→T*F
T→F
F→(E)
F→id

الف- ۱۱　　ب- ۱۲

ج- ۱۳　　د- ۹

۹۷- کدامیک از موارد ذیل صحیح است.

الف- هر گرامر LL(1) یک گرامر LR(1) است.

ب- هر گرامر LR(1) یک گرامر LL(1) است.

ج- تجزیه کننده های بالا به پایین درخت تجزیه را به ترتیب inorder می سازند.

د- تجزیه کننده های بالا به پایین درخت تجزیه را به ترتیب postorder می سازند.

۹۸- با توجه به برنامه ذیل اگر رشته ورودی 2*4+1 باشد خروجی کدام گزینه است.

```
%{
 #include <math.h>
 #include <conio.h>
%}

%token  NUM
%left '*' '/'
%left '+' '-'
%right '^'
```

```
%%
input:  exp { printf("\t%d\n", $1); }   ;

exp:   NUM       { $$ = $1; }
       | exp '+' exp       { $$ = $1 + $3; }
       | exp '-' exp       { $$ = $1 - $3; }
       | exp '*' exp       { $$ = $1 * $3; }
       | exp '/' exp       { $$ = $1 / $3; }
       | exp '^' exp       { $$ = pow ($1, $3); }
       | '(' exp ')'       { $$ = $2;}
;
%%
main(){
yyparse();
}
yyerror(){}
yylex(){

char ch;
int  sum=0;
ch=getc(stdin);
while(ch==' ') ch=getc(stdin);
if(ch=='+' || ch=='-' || ch=='*' || ch=='/' || ch=='^' || ch==')' || ch== '(')
                return(ch);
while(ch>='0' && ch<='9'){
   sum=sum*10 + (ch-'0');
   ch=getc(stdin);
}
ungetc (ch,stdin);
yylval=sum;
return(NUM);
}
```

الف- ۱۰ ب- ۹

ج- ۷ د- ۵

۹۹- کدام یک از گرامر های ذیل از نوع عملگر است.

الف- $A \rightarrow Bb|cBd|ed|cba$

$B \rightarrow \epsilon$

ب- $A \rightarrow BbBb|\ CcCb$

$B \rightarrow \epsilon$

$C \rightarrow \epsilon$

ج- $A \rightarrow Bb|cBd|ed|cba$

$B \rightarrow d$

د- هیچکدام

۱۰۰- کدام گزینه در مورد گرامر ذیل صحیح است.

$A \rightarrow Bb|cBd|ed|cba$
$B \rightarrow e$

الف- این گرامر SLR(1) نیست.

ب- این گرامر LR(0) است.

ج- این گرامر از نوع عملگر نیست

د-هیچکدام

کلید سوالات تستی

	الف	ب	ج	د		الف	ب	ج	د		الف	ب	ج	د		الف	ب	ج	د	
۱		√			۲۶			√		۵۱		√			۷۶	√				
۲	√				۲۷				√	۵۲	√				۷۷	√				
۳	√				۲۸			√		۵۳		√			۷۸			√		
۴			√		۲۹				√	۵۴	√				۷۹		√			
۵		√			۳۰			√		۵۵		√			۸۰					√
۶		√			۳۱	√				۵۶		√			۸۱	√				
۷		√			۳۲	√				۵۷		√			۸۲					√
۸				√	۳۳	√				۵۸	√				۸۳		√			
۹		√			۳۴	√				۵۹				√	۸۴	√				
۱۰		√			۳۵				√	۶۰				√	۸۵					√
۱۱				√	۳۶		√			۶۱			√		۸۶					√
۱۲				√	۳۷			√		۶۲		√			۸۷		√			
۱۳	√				۳۸	√				۶۳				√	۸۸					√
۱۴			√		۳۹	√				۶۴	√				۸۹					√
۱۵	√				۴۰				√	۶۵			√		۹۰					√
۱۶	√				۴۱				√	۶۶				√	۹۱	√				
۱۷				√	۴۲	√				۶۷			√		۹۲		√			
۱۸		√			۴۳		√			۶۸				√	۹۳			√		
۱۹				√	۴۴	√				۶۹	√				۹۴			√		
۲۰	√				۴۵		√			۷۰				√	۹۵					√
۲۱				√	۴۶	√				۷۱				√	۹۶		√			
۲۲	√				۴۷			√		۶۲			√		۹۷	√				
۲۳				√	۴۸	√				۷۳	√				۹۸	√				
۲۴				√	۴۹	√				۷۴				√	۹۹			√		
۲۵	√				۵۰	√				۷۵			√		۱۰۰	√				

خوانندهٔ محترم

این پرسشنامه به منظور ارتقای کیفیت کتابهای درسی و رفع نواقص آنها تهیه شده است. دقت شما در پاسخگویی به این پرسشنامه در پایان هر نیمسال ما را در تحقق این هدف یاری خواهد کرد.

نام کتاب ..... نام مؤلف/مترجم ..... سال انتشار .....

وضعیت پاسخگو: عضوعلمی پیام‌نور☐عضوعلمی سایردانشگاهها☐رشتهٔتخصصی ..... سابقهٔتدریس .....

دانشجوی‌پیام‌نور☐ دانشجوی سایردانشگاهها☐ رشتهٔتحصیلی ..... ورودی سال .....

سؤال	خیلی زیاد	زیاد	کم	اصلاً کم
۱. آیا از زمان تحویل و نحوهٔ دسترسی به کتاب راضی بودید؟				
۲. آیا حجم کتاب با توجه به تعداد واحد مناسب بود؟				
۳. آیا راهنماییهایی لازم برای مطالعهٔ کتاب منظور شده بود؟				
۴. آیا در ترتیب مطالب کتاب سلسله مراتب شناختی (آسان به مشکل) رعایت شده بود؟				
۵. آیا تقسیم‌بندی مطالب در فصلها و یا بخشها متناسب و بجا بود؟				
۶. آیا متن کتاب روان و ساده و جمله‌ها قابل‌فهم بود؟				
۷. آیا به روز بودن مطالب و آمارها رعایت شده بود؟				
۸. آیا مطالب تکراری داشت؟				
۹. آیا پیوستگی مطالب با درسهای پیش‌نیاز رعایت شده بود؟				
۱۰. آیا مثالها، شکلها، نمودارها، جدولها و...گویا بودند و در فهم مطلب تأثیر داشتند؟				
۱۱. مطالعهٔ هدفهای کلی، آموزشی/رفتاری تا چه اندازه به درک بهتر شما کمک کرد؟				
۱۲. آیا خودآزماییهای کتاب به گونه‌ای بود که تمام مطالب درسی را شامل شود؟				
۱۳. آیا پاسخ خودآزماییها و تمرینها کامل و گویا بود؟				
۱۴. چقدر با غلطهای املایی و اشکالهای چاپی مواجه شدید؟				
۱۵. کیفیت چاپ و صحافی کتاب چگونه بود؟				
۱۶. آیا طرح روی جلد کتاب مناسب بود؟				
۱۷. چنانچه از وسایل کمک‌آموزشی از قبیل نوار، فیلم، لوح فشرده و... استفاده کرده‌اید، آیا به درک بهتر شما کمک کرده است؟				
۱۸. تا چه اندازه این کتاب شما را از حضور در کلاس بی‌نیاز کرد؟				

لطفاً چنانچه با اشکالهای تایپی یا محتوایی و مطالب تکراری مواجه شده‌اید، فهرستی از آنها را با ذکر شمارهٔ صفحه ضمیمه کنید.

در مجموع کتاب را چگونه ارزیابی می‌کنید؟ عالی ☐ خوب ☐ متوسط ☐ ضعیف☐

در صورت تمایل سایر پیشنهادهای خود را نیز بنویسید.

این پرسشنامه را پس از تکمیل از کتاب جدا کنید و به قسمت آموزش مرکز تحویل دهید یا مستقیماً به نشانی تهران ۱۹۵۶۹- صندوق پستی ۴۶۹۷-۱۹۳۹۵، مدیریت تدوین ارسال فرمایید.

با تشکر

مدیریت تدوین

دانشگاه پیام نور

دانشگاه پیام نور ۱۱۷۳

گروه کامپیوتر (۱۲/د)

ISBN:978-964-387-196-3