مراجع درس : ۴
مقدمه : ۴
در این درس : ۴
مطالب این درس بصورت شماتیک : ۴
فصل اول : مقدمات و تعاریف ۵
کامپایلر : ۵
تاریخچه : ۵
اجزاء سیستم پردازش زبان : ۶
اجزاء کامپایلر (یا فازهای آن) : ۶
گذر یا Pass : ۸
مروری بر درس نظریهٔ زبان‌ها و ماشین‌ها : ۸
نمادهای گرامر : ۸
تقسیمات زبان‌ها : ۸
زبان‌های منظم یا Regular : ۸
زبان‌های مستقل از متن : ۹
زبان‌های وابسته به متن : ۹
فصل دوم : فاز تحلیل لغوی یا Lexical Analysis **و یا** Scanner ۱۰
وظایف تحلیل‌گر لغوی : ۱۰
انواع Tokenها : ۱۰
الگوی Tokenها : ۱۰
الگوی تشخیص ID : ۱۱
الگوی تشخیص Keyword : ۱۱
الگوی تشخیص NUM : ۱۲
الگوی تشخیص عملگرهای مقایسه‌ای : ۱۲
الگوی تشخیص رشته‌ها : ۱۲
تفاوت‌های DFA و NFA : ۱۲
تبدیل عبارات منظم به NFA : ۱۳
تعریف تابع $\lambda-Closure$: ۱۳
تبدیل NFA به DFA : ۱۳
بهینه سازی DFA : ۱۴
روش‌های ایجاد یا پیاده سازی تحلیل‌گر لغوی : ۱۵
ساختار زبان LEX : ۱۵
فصل سوم : فاز تحلیل‌گر نحوی ۱۶
دلایل استفاده از گرامرهای مستقل از متن : ۱۶
نمایش ساختار دستورات با گرامر مستقل از متن : ۱۶
نقش تحلیل‌گر نحوی در کامپایلر : ۱۶

روش‌های ساخت درخت تجزیه : ۱۷
روش‌های استاندارد تجزیه : ۱۷
روش‌های بالا به پائین یا بازگشتی کاهشی (Recursive Descent) : ۱۷
روش‌های پائین به بالا یا Shift Reduce : ۱۷
اشتقاق : ۱۷
اشتقاق Right Most و Left Most : ۱۷
ابهام در گرامر : ۱۸
چپ گردی یا Left Recursive : ۱۸
فاکتور چپ : ۱۹
مروری بر تحلیل‌گر نموی : ۱۹
نمودهٔ عملکرد تجزیهٔ بالا به پائین یا اشتقاق : ۱۹
تجزیه کننده‌های پیش‌نگر : ۲۰
پیاده سازی تجزیه کنندهٔ پیش‌نگر : ۲۰
ساختار جدول تجزیهٔ M : ۲۰
عملکرد تجزیه کنندهٔ پیش‌نگر : ۲۱
روش ساختن جدول تجزیهٔ M : ۲۲
تابع First() : ۲۲
تابع Fallow() : ۲۲
تشخیص گرامرهای LL(1) بدون رسم جدول تجزیهٔ M : ۲۴
مفهوم اصلاح خطا یا Error Recovery : ۲۴
روش‌های اصلاح خطا برای تجزیهٔ پیش‌نگر : ۱- روش اضطراری یا Panic Mode : ۲۴
مجموعهٔ هماهنگ (S یا Synchronize) : ۲۵
الگوریتم اصلاح خطا در روش Panic Mode : ۲۵
الگوریتم اصلاح خطا در روش Phraze Level : ۲۵
تجزیه کنندگان پائین به بالا : ۲۶
تداخل‌ها در تجزیهٔ پائین به بالا : ۲۷
تجزیهٔ اولویتْ عملگر یا OP : ۲۷
جدول روابط تقدمی : ۲۸
الگوریتم تجزیه به روش OP : ۲۸
ایجاد جدول روابط تقدمی : ۲۸
تابع FirstTerm() : ۲۸
تابع LastTerm() : ۲۸
الگوریتم ایجاد جدول تقدمی : ۲۹
روش‌های اصلاح خطا برای تجزیهٔ اولویتْ عملگر یا OP : ۳۰
روش اضطراری یا Panic Mode : ۳۰
روش Phraze Level : ۳۰

تجزیه به روش تقدم ساده یا SP : ۳۱
ایجاد جدول روابط تقدمی در تجزیهٔ تقدم ساده : ۳۱
الگوریتم ایجاد جدول روابط تقدمی در تجزیهٔ تقدم ساده : ۳۱
الگوریتم تجزیه به روش تقدمی ساده : ۳۲
روش‌های اصلاح خطا برای تجزیهٔ تقدمی ساده : ۳۲
تجزیه به روش LR : ۳۲
عملکرد تجزیه کننده‌های LR : ۳۲
الگوریتم تجزیه به روش LR : ۳۳
ایجاد جدول تجزیهٔ LR به روش SLR : ۳۴
گرامر SLR(1) : ۳۴
رسم دیاگرام SLR : ۳۵
ایجاد جدول تجزیه از روی دیاگرام SLR : ۳۶
رسم نمودار تبدیل وضعیت CLR : ۳۸
گرامر CLR(1) : ۳۹
نمودهٔ ایجاد دیاگرام LALR : ۳۹
مقایسهٔ جداول SLR ، CLR و LALR : ۴۱
خلاصه‌ای از فصل تحلیل نحوی : ۴۱
فصل چهارم : فاز تحلیل معنایی و تولید کد میانی ۴۲
انبارهٔ محدوده یا Scope Stack : ۴۲
تولید کد میانی یا کدهای سه آدرسه : ۴۲
تولید کد میانی به روش بالا به پائین : ۴۳
تولید کد میانی مربوط به عبارات جبری و دستور انتساب : ۴۳
عملیات pid یا pid Semantic Action : ۴۳
عملیات add یا جمع : ۴۴
عملیات mul یا ضرب : ۴۴
عملیات assign یا انتساب : ۴۴

اصول طراحی کامپایلرها

مراجع درس :

۱. اصول طراحی کامپایلرها
۲. کامپایلرها ، اصول و مفاهیم

مؤلف : اولمن v2007 (توجه داشته باشید که این مرجع است و نه منبع (یعنی آموزشی نیست))

مقدمه :

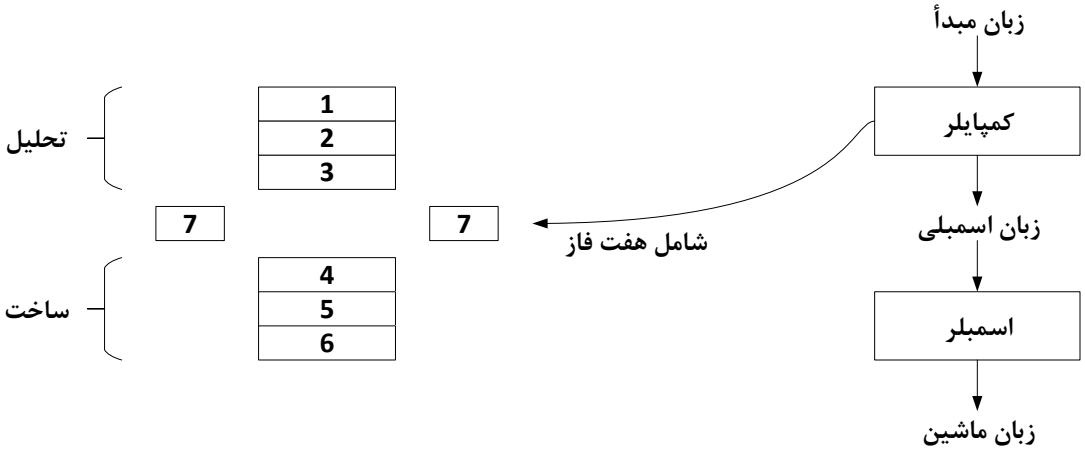
درس‌های این گرایش (نرم افزار) به دو دسته کلی تقسیم می‌شوند :

۱. درس‌های اپلیکیشنی و کاربردی : مانند دروس برنامه نویسی
۲. درس‌های مربوط به علوم کامپیوتر : مانند سیستم عامل و از جمله همین درس : این دروس کاملاً تئوری هستند

در این درس :

- خواهیم فهمید که یک زبان برنامه نویسی چطور کار می‌کند
- این درس شامل چندین الگوریتم است که تا حد زیادی به هم شبیه هستند ؛ در نتیجه بایستی خوب آنها را بفهمید
- در پایان ترم نیز قادر خواهیم بود که یک کامپایلر را کاملاً بشناسیم

مطالب این درس بصورت شماتیک :



- فصل اول : مفاهیم ابتدایی (خلاصه‌ای از کل کتاب)
- فصل دوم : (فصل سوم از کتاب) (فاز یکم) تحلیل لغوی : آیا لغت نوشته شده ، درست نوشته شده یا خیر ؛ آیا لغت معنی می‌دهد یا خیر. هر چیزی که در این فصل نیاز است ، در درس نظریهٔ زبان خوانده شده.
- فصل سوم : (فاز دوم) تحلیل نحوی : آیا جای کلمهٔ نوشته شده ، درست است یا خیر (بیشترین زمان را خواهد گرفت)
- فصل چهارم : (فاز سوم و چهارم) تحلیل معنایی و ایجاد کد میانی : برای مثال یک مقدار رشته‌ای به یک متغیر عددی انتساب نشده باشد. کد میانی نیز دستوراتی شبیه به دستورات اسمبلی است که راحت‌تر به این زبان تبدیل می‌شوند
- فصل پنجم و ششم : (فاز پنجم و ششم) که مربوط به درس کامپایلر پیشرفته است.

- میان ترم تا پایان فصل دوم و شش نمره خواهد داشت
- پایان ترم نیز چهارده نمره خواهد داشت

اصول طراحی کامپایلرها

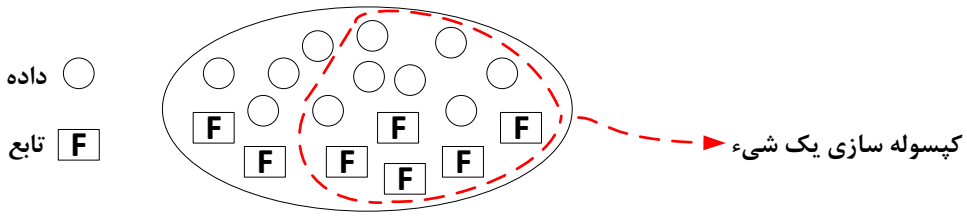
« فصل اول : مقدمات و تعاریف »

کامپایلر :

که به معنای مترجم و بخشی از یک زبان برنامه نویسی است و به آن سیستم پردازش زبان نیز گفته می‌شود. مترجم زبان مبدأ به زبان مقصد

تاریخچه :

- کامپیوترهای ابتدایی کامپایلر نداشت. ورودی آن مستقیماً به زبان ماشین ، یعنی صفر و یک بود که بوسیلهٔ یکسری کلید کنترل میشد و خروجی آن یکسری لامپ بود. از این کامپیوترها برای حل کردن دستگاه‌های معادلاتی کمک گرفته میشد.
- ۱۹۵۰ : ورود اولین کامپایلر برای اولین زبان برنامه نویسی ، یعنی فرتـرن ؛ (فرتـرن یعنی ترجمهٔ فرمول به زبان ماشین). در زبان‌های اولیه ، یکسری دستور وجود داشت که با شمارهٔ خط کنترل و به ترتیب همین شمارهٔ خطوط بود که اجـرا میشد. چون این زبان‌ها هیچ قـاعده و قانونی نداشتند ، اگـر تعداد خطوط بیش از پنج یا شش هزار خط میشد ، برنامه دچار آشفتگی میشد. به این زبان‌ها ، زبان‌های اسپاگتی (یک کلاف سر در گم) نیز گفته می‌شود. مانند Cobol ، GW Basic
- ۱۹۶۰ : ورود زبان‌های ساختیافته : برنامه در این زبان‌ها دارای ساختار خاصی بود. خطوط دستورات دارای شماره نبود ولی هر دستور جای مشخصی داشت و نمی‌توانستی هر دستور را هرجایی بنویسی. دستورات در این برنامه‌ها از بالا به پایین اجرا میشد. دستورات پرشی در این زبان‌ها قابل استفاده شد ولی استفاده از آنها پیشنهاد نمیشد. ویژگی این زبان‌ها ، امکان استفاده از توابع یا زیر برنامه‌ها بود و می‌توانست بین ۶۰ تا ۷۰ هزار خط برنامه را در خود جای دهد ولی باز هم دارای محدودیت بود.
- ۱۹۷۰ : ورود زبان‌های شیءگرا : شیء عبارت است از یکسری داده و عملیات‌های مرتبط به هم که در یک قالب نگهداری می‌شود. در این زبان‌ها ، تنها یک شیء بواسطهٔ یک تابع می‌تواند به داده‌های اشیاء دیگر دسترسی پیدا کند و این منجر به مدیریت بهتر خواهد شد. در این زبان‌ها ، برنامه می‌تواند خیلی خیلی بزرگ و پیچیده باشد.



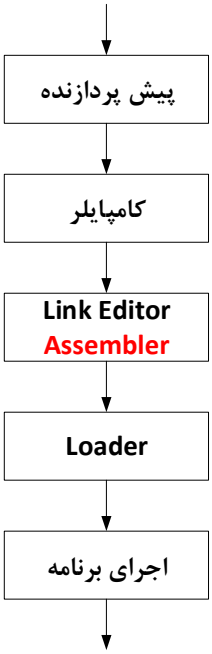
- ۱۹۸۰ : ورود زبان‌های ویژوال یا بصری : زبان‌های شیءگرا سخت بود بخصوص برای طراحی ظاهر برنامه. با ورود این زبان‌ها ، طراحی ظاهر برنامه بسیار ساده شد.
- ۱۹۹۰ : ورود زبان‌های قابل حمل یا Portable : در زبان‌های قبلی ، فایل تولید شده به محیط و سیستم عامل بستگی داشت ولی در این نوع زبان‌ها ، خروجی ایجاد شده وابسته به یک ساختار خاص نبوده و روی ساختارهای دیگر هم اجرا می‌شود. زبان‌های برنامه سازی تحت وب جزء این دسته از زبان‌ها هستند.

رایانه‌ها به دو دستهٔ کلی زیر تقسیم بندی می‌شوند :

- سازگار با IBM
- سازگار با Apple

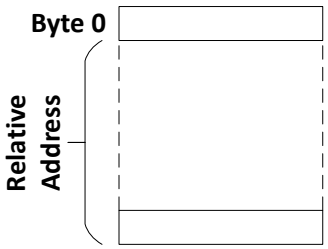
توجه داشته باشید که در تمامی زبان‌های فوق ، ساختار اصلی کامپایلرها ، هیچ تفاوتی با یکدیگر ندارند

اجزاء سیستم پردازش زبان :

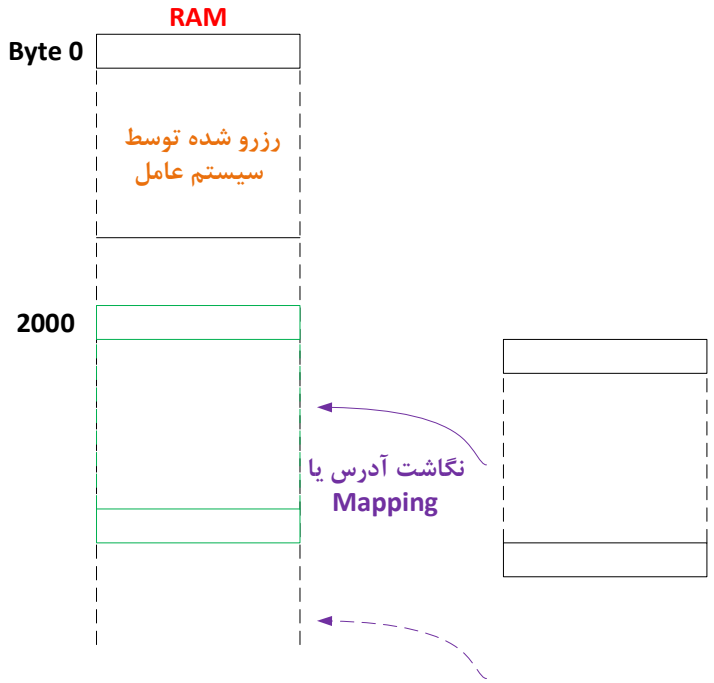


۱. پیش پردازنده : وظیفهٔ این مرحله ، یکپارچه کردن کلیهٔ فایل‌های موجود در پروژه در قالب یک فایل است. پس از آماده شدن ، فایل به کامپایلر ارسال خواهد شد.
خروجی این مرحله ، فایل مبدأ بوده و به زبان سطح بالاست. یعنی به زبان همان محیطی که برنامه در آن نوشته شده است و ما با آن سر و کار داریم.

۲. کامپایلر : وظیفهٔ این مرحله ، ترجمهٔ فایل آماده شده در مرحلهٔ قبل ، به زبان ماشین است.
خروجی این مرحله ، فایل مقصد بوده و به زبان ماشین (یعنی زبان صفر و یک) می‌باشد. این فایل حاوی آدرس‌های نسبی (آدرس‌های قابل حمل) است. در واقع بقیهٔ آدرس‌ها ، نسبت به "صفر" محاسبه می‌شوند.



فایل‌های exe ، همواره از آدرس صفر شروع می‌شوند. معمولاً خروجی کامپایلرها ، فایل‌های exe هستند.
فایل‌های Com ، از یک آدرس خاص شروع می‌شوند. این فایل‌ها ، با اسمبلی ساخته می‌شوند.

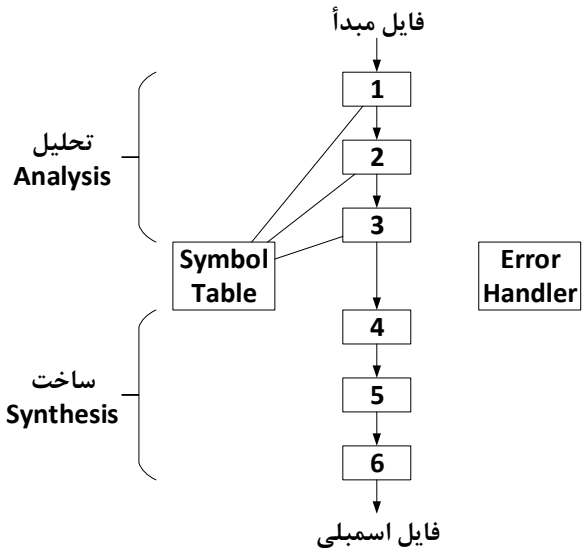


۳. Link Editor : گاهی یک فایل exe ، با فایل‌های دیگری نیز همراه است (برای مثال یک فایل dll). در اینصورت خروجی اسمبلی نیز چند تکه خواهد شد. وظیفهٔ این مرحله ، نگه داشتن این فایل‌های تکه تکه است که در Ram قرار گرفته‌اند.

۴. Loader : در واقع این قسمت ، بخشی از سیستم عامل است. وظیفهٔ این مرحله ، بارگذاری فایل اصلی (به زبان اسمبلی) در حافظه است. یا به زبان دیگر ، انتقال فایل اجرایی از حافظهٔ Passive یا جانبی ، به حافظهٔ اصلی یا Ram برای اجراست. مهمترین عمل در این مرحله ، نگاشت آدرس یا Mapping است.

نکته :
- آدرس ۲۰۰۰ ، یک آدرس فیزیکی است.
- سیستم رم ، بر اساس سیستم تورینگ است.

اجزاء کامپایلر (یا فازهای آن) :



۱. تحلیل لغوی یا Lexical Analysis و یا Scanner : در این مرحله ، تک تک لغات برنامه ، برای درست بودن بررسی می‌شوند. بررسی می‌شود که هر لغت چیست و نوع آنرا شناسایی می‌کند.
روش کار به اینصورت است که از حرف اول هر خط شروع به خواندن می‌کند تا به یک delimiters (حروف جدا کننده) برسد.

کد برنامه	لغات شناسایی شده

`if (A>=B) then`
`  A:=B*150`
`else`
`  Writeln("Error");`
	`if,(,A,>=,B,),then`
	`A,:=,B,*,150`
	`else,`
	`Writeln,(,"Error",),;,`

کامپایلر برای انجام این مرحله ، از زبان‌های منظم استفاده می‌کند. ویژگی اینگونه زبان‌ها در این است که الگوی یک کلمه را به خوبی نشان می‌دهد.
برای نمایش زبان‌های منظم ، از DFA و یا NFA و یا بطور کلی از FA استفاده می‌کنیم که متناهی نیز هستند.

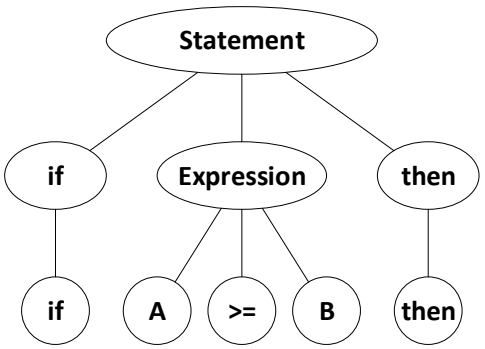
خروجی این مرحله ، آدرس خود Token ها هستند. به هر لغت در کامپایلر ، Token گفته می‌شود که از یکسری حروف تشکیل شده و دارای معنی نیز هست. به خروجی ، Token Value نیز گفته می‌شود. همانطوری که قبلاً هم به آن اشاره شد ، علاوه بر بررسی لغات ، نوع آنها نیز بررسی می‌شود. پس Tokenها ، نوع‌هایی هم دارند که به آنها Token Type گفته می‌شود.

اطلاعات فوق در جدول نمادها یا همان Symbol Table ریخته می‌شود. در واقع این جدول ، یک ماتریس است که هر سطر آن یک Token ، Token Type ، Token Value و یکسری چیزهای دیگری منجمله آدرس Token شناسایی شده است که به فازهای بعدی ارسال خواهد شد.

در واقع در این مرحله ، زبانی به نام Lexical وجود دارد که خروجی Lexical Analysis در جدول نمادها ریخته می‌شود.

در صورتیکه در این مرحله به خطایی برخورد کند ، با استفاده از Error Handler ، آن خط را نشانه گذاری کرده (یا آنرا رنگی می‌کند) و خطای مناسبی چاپ خواهد کرد.

۲. تحلیل نحوی یا Synatx Analyser و یا Parser : Syntax به معنای عمومی ترتیب بوده و در واقع در این مرحله ، درستی ترتیب Token ها بررسی می‌شوند.



کامپایلر برای انجام این مرحله ، از زبان‌های مستقل از متن استفاده می‌کند که در پیاده سازی این زبان‌ها از PDA کمک می‌گیریم.

وظیفهٔ این مرحله ، ایجاد یک درخت تجزیه با توجه به گرامر دستور مورد نظر است که با کمک زبانی به نام yacc که در کامپایلر وجود دارد ، انجام می‌شود و آنرا به فازهای بعدی ارسال خواهد کرد.

در صورتیکه در این مرحله نیز به خطایی برخورد کند ، با استفاده از Error Handler خطای مناسبی چاپ خواهد شد.

۳. تحلیل معنایی یا Semantic Analyser : در این مرحله ، معنا بررسی می‌شود.

برای مثال اگر خطوط زیر قبل از تکه کدی که در فاز یک نوشته است ، وجود داشته باشد ، دستورات دارای معنی نبوده و خطایی چاپ خواهد شد.

```
int A;
char B;
```

خروجی در این مرحله ، همان درخت تجزیهٔ تولید شده در مرحلهٔ قبل است.

۴. تولید کد میانی یا Intermediate Code Generator : همانطور که قبلاً به آن اشاره شد ، کدهای میانی راحت‌تر و ساده‌تر به کدهای اسمبلی تبدیل می‌شوند. این فاز وظیفهٔ خود را بوسیلهٔ شکستن دستورات به دستورات کوچکتر انجام می‌دهد. برای مثال داریم :

$$A := B + C * D \quad \Rightarrow \quad Declare\ Variable\ T1, T2 \quad \Rightarrow \quad \begin{cases} T1 := C * D \\ T2 := B + T1 \\ A := T2 \end{cases}$$

به این دستورات ساده شده ، اصطلاحاً کدهای سه آدرسه گفته می‌شود. دلیل آن این است که هر خط دستور ، حداکثر حاوی سه متغیر است. برای مثال در خطوط فوق ، دو دستور اول حاوی سه متغیر و دستور آخر حاوی دو متغیر است. خروجی در این فاز ، همین کدهای سه آدرسه هستند.

اطلاعات مورد نظر در این فاز ، از جدول نمادها گرفته شده و در صورتیکه در این مرحله نیز به خطایی برخورد کند ، با استفاده از Error Handler خطای مناسب را نشان خواهد داد.

۵. بهینه سازی یا Optimizer : کدهای سه آدرسهٔ تولید شده در مرحله قبل ، بهینه سازی می‌شوند. برای مثال از سه دستور تولید شده در مثال فوق ، دستورات را می‌توان بصورت زیر بازنویسی کرد (حذف دستور آخر و تغییر متغیر تعریف شدهٔ دوم در دستور دوم) :

$$\begin{cases} T1 := C * D \\ A := B + T1 \end{cases}$$

۶. تولید کد نهایی : از خروجی کدهای بهینه شده ، فایل اسمبلی در این مرحله ساخته خواهد شد.

نکته :

کامپایلری که خروجی آن ، فایل اسمبلی است ، کد غیر قابل حمل تولید کرده و خروجی آن مخصوص ماشینی است که کامپایلر روی آن وجود دارد. در صورتیکه خروجی را از فاز چهارم بگیریم ، کد قابل حمل در اختیار خواهیم داشت ؛ زیرا به فایل اسمبلی مخصوص آن ماشین تبدیل نشده است. زبان Java و Net. خروجی خود را از فاز چهارم می‌گیرند و لذا برای اجرا آن نیاز به ابزاری است که دو مرحلهٔ آخر را در خود جای داده باشد و بتواند کد میانی را به فایل اجرایی تبدیل کند. این ابزار در Java ، Java Virtual Machine یا به اختصار JVM و در دات نت ، Framework نامیده می‌شود.

اصول طراحی کامپایلرها

گذر یا Pass :

هر بار که کامپایلر ، برنامه را از ابتدا تا انتها می‌خواند و کاری انجام می‌دهد ، یک گذر یا Pass نامیده می‌شود. با توجه به شش فاز فوق ، قاعدتاً نیاز به شش گذر خواهد بود ولی یکسری از فازها قابلیت انجام در یک فاز را بصورت مشترک دارند. سه فاز اول در یک گذر و سه فاز دوم در گذر دوم انجام می‌گیرد.
به این کامپایلرها ، اصطلاحاً کامپایلرهای دو گذره گفته می‌شود.

برای چگونگی عملکرد فازها ، نیاز به یکسری دانش قبلی از درس نظریهٔ زبان‌ها و ماشین‌ها است. در ادامه مروری می‌کنیم بر این درس :

مروری بر درس نظریهٔ زبان‌ها و ماشین‌ها :

الفبا : مجموعه‌ای با پایان یا متناهی از نشانه‌ها و سمبول‌ها که با Σ نشان داده می‌شود.
رشته : هر ترکیبی از الفبا در کنار هم ، یک رشته نامیده می‌شود و با W نشان داده می‌شود.
زبان : زیر مجموعه‌ای از رشته‌های تولید شده از یک الفبا که مسلماً برای ما دارای معنی نیز باشند ، تشکیل یک زبان می‌دهند که با L نشان داده می‌شود.
گرامر (قوانین یا الگوهای تولید) : در واقع قواعدی هستند که رشته‌های آن زبان را تولید می‌کنند. این قواعد را با P و گرامر را با G نشان می‌دهند.

نمادهای گرامر :

۱. حروف کوچک : نشاندهندهٔ الفبا هستند که به آنها متغیرهای پایانی یا ترمینال گفته می‌شوند مثل a,b
۲. حروف کوچک آخر : نشاندهندهٔ رشته هستند مثل w,x,y,z . رشته‌ای مثل $w=abcd$
۳. حروف بزرگ : نشاندهندهٔ متغیرها یا ناپایانی‌ها هستند مثل A,B
۴. حروف بزرگ آخر : نشاندهندهٔ یک یا چند متغیر است مثل X,Y,Z که در آن داریم $X=ABC$
۵. حروف یونانی : نشاندهندهٔ یک یا چند متغیر و یا نماد الفبا است مثل α,β

تقسیمات زبان‌ها :

۱. زبان‌های منظم یا Regular
۲. زبان‌های مستقل از متن
۳. زبان‌های وابسته به متن
۴. زبان‌های بدون قاعده : این زبان‌ها در کامپایلر استفاده نمی‌شوند.

زبان‌های منظم یا Regular :

اگر هر نماد از زبان فقط به تکرار خودش وابسته باشد ، آن زبان منظم است. یا به عبارت دیگر تعداد متناهی از تکرار یک نماد یا چند نماد در کنار هم قرار گیرند. مانند a^n و یا a^nb^m که از شکل رشته می‌توان به منظم بودن آن پی برد.

این زبان‌ها به دو صورت قابل نمایش هستند :

۱. گرامر منظم : که همیشه یک الگوی ثابت دارند. $A\to aX\mid a$ یا $A\to Xa\mid a$
برای مثال ، گرامر زیر نشاندهندهٔ یک زبان منظم نیست :

$$S\to aS\mid A$$
$$A\to Ab\mid b$$

۲. عبارات منظم : که روش ساده‌تری است و شامل سه عملگر $Concat(.)$ ، $Or(|)$ و $Star(*)$ می‌باشد.

$a^*\to a^0,a^1,a^2,\ldots\to\lambda,a,aa,\ldots$ $\quad a\mid b\to a,b$ $\quad a\,.\,b\to ab$

پیاده سازی این زبان‌ها توسط FA بود (DFA و یا NFA) که ماشین‌های خودکار متناهی بودند که هیچگونه حافظه‌ای ندارند.

تحلیل لغوی از زبان‌های منظم استفاده می‌کند

نکته :

- تعداد تکرار ثابت ، همواره زبان منظم می‌دهد مثل $a^{1000}b^{1000}$
- اگر در توان ، تابعی وجود داشته باشد ، قطعاً منظم نیست مثل $a^{n!}$ که جزء زبان‌های بی قاعده است.

زبان‌های مستقل از متن :

در اینصورت رشته از تعداد نامتناهی از حداکثر دو نمادی که به هم وابسته هستند تشکیل شده است. مانند a^nb^n که در آن تعداد a,b به هم وابسته است یا a^nb^m که در آن شرطی بین n,m باشد مثل $n > m \quad Or \quad n \neq m$ یا $a^nb^mc^n$ که در آن تعداد a,c به هم وابسته است یا $a^nb^mc^md^n$ که در آن تعداد b,c و همچنین a,d به هم وابسته است.

این زبان‌ها تنها با گرامر قابل نمایش هستند و شکل کلی آن بصورت $A \to \alpha$ است (در سمت چپ تنها یک متغیر)

پیاده سازی آنها نیز با PDA است که همان ماشین‌های FA هستند با این تفاوت که در آنها Stack بکار رفته است

تحلیل نحوی از زبان‌های مستقل از متن استفاده می‌کند.

زبان‌های وابسته به متن :

این زبان‌ها را با PDA نمی‌توان پیاده سازی کرد. در بعضی از آنها هم وابستگی‌هایی وجود دارد که همدیگر را قطع می‌کنند.

مثل $a^nb^mc^n$ که نمی‌توان آنرا با PDA پیاده کرد و $a^nb^mc^nd^m$ که در آن تعداد b,d و همچنین a,c به هم وابسته است ولی همدیگر را قطع می‌کنند.

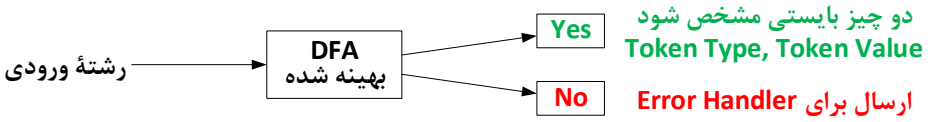
این زبان‌ها نیز تنها با گرامر قابل نمایش هستند و شکل کلی آن بصورت $\alpha \to \beta$ است (نه در سمت چپ و نه در سمت راست محدودیتی وجود ندارد)

« فصل دوم : فاز تحلیل لغوی یا Lexical Analysis و یا Scanner »

در این فصل تنها با زبان‌های منظم سر و کار داریم. وظیفهٔ این فاز ، تشخیص Tokenها است. برای این منظور ، بایستی برای هر Token ، یک الگو مشخص کرد و این الگو ، یک عبارت منظم است که بهترین ابزار برای نمایش الگوهاست.

بطور کلی در این فصل بایستی مراحل زیر طی شود :

۱. مشخص کردن الگو برای هر Token
۲. پیاده سازی الگو با تبدیل الگو به NFA (که بسیار ساده‌تر از DFA است). برای این منظور از الگوریتم آن استفاده می‌کنیم
۳. چون کامپایلر نمی‌تواند از NFA استفاده کند ، پس آنرا به DFA تبدیل می‌کنیم. برای این منظور نیز الگوریتم وجود دارد
۴. DFA به دست آمده را بهینه می‌کنیم. هر چه یک DFA بهینه‌تر باشد ، الگوها را راحت‌تر شناسایی می‌کند.



نکته :

- رشتهٔ ورودی در شکل فوق ، در حقیقت فایلی است که در مرحلهٔ پیش پردازنده تولید شده است.

وظایف تحلیل‌گر لغوی :

۱. شناسایی و تشخیص Tokenها
۲. حذف Commentها و فضاهای سفید (White Space (WS) که شامل Enterها و Spaceهای اضافی است)
۳. شماره گذاری خطوط برنامه برای پیغام خطا (در پیغام خطا ، مشخص شده که در خط شمارهٔ ... ، خطا وجود دارد)

انواع Tokenها :

۱. کلمات کلیدی یا رزرو شده (Keyword) : شامل تمامی دستورات یک زبان برنامه نویسی. مانند : `if,then,else,Writeln`
۲. ID یا Identifier یا شناسه : شامل تمامی اسامی که در طول یک برنامه توسط کاربر تعریف می‌شود. (مانند متغیرها ، نام توابع و ...). مانند : `A,B`
۳. اعداد یا Number (به اختصار Num). مانند : `150`
۴. عملگرها یا Operators (به اختصار OP). مانند : `*,>=,:=`
۵. رشته‌ها یا String. مانند : `"Error"`
۶. علائم. مانند : `;,(,)`

الگوی Tokenها :

حال که در مرحلهٔ قبل ، انواع Token را شناختیم ، بایستی برای آن الگو ارائه کنیم. برای بیان این الگو ، از گرامر منظم و یا عبارات منظم می‌توانیم استفاده کنیم که بیان گرامر به مراتب سخت‌تر از بیان عبارت منظم است. برای مثال داریم :

$$L_1 = \{a^n b^m \quad | \quad n,m \geq 0\}$$

گرامر بصورت $G(T,V,S,P)$ تعریف می‌شود : که در آن T همان ترمینال و V همان متغیرها هستند و داریم $T = \{a,b\}$

برای نوشتن گرامر ، بهتر است تعدادی از رشته‌های آنرا بنویسیم. داریم :

$$\lambda, a, aa, ..., b, bb, ..., ab, aabb, ..., aab, abb, ...$$

پس خواهیم داشت :

$$S \rightarrow AB$$
$$A \rightarrow aA \mid \lambda$$
$$B \rightarrow bB \mid \lambda$$

ولی با استفاده از عبارات منظم خواهیم داشت : $a^* b^*$. که البته این برای یک رشته و زبان فوق العاده ساده بود.

مثال : برای الگوی زبان زیر یک عبارت منظم بنویسید :

$$L_2 = \{aaWb \quad : \quad W \in \Sigma^*\} \quad , \quad \Sigma = \{a,b\}$$
$$= aa(a \mid b)^* b$$

الگوی تشخیص ID :

در یک زبان ، ابتدا بایستی کلیهٔ IDها جمع آوری شده و سپس برای آن یک الگو ارائه کرد. حال فرض کنید زبان یک ID بصورت زیر باشد :

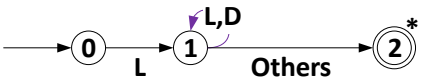
$L_{ID} = \{$ شامل کلیهٔ اعداد و حروف که حتماً با یک حرف شروع شود $\}$

اعداد : $0|1|2|3|...|9 \rightarrow D$ حروف : $a|b|b|...|z|A|B|C|...|Z \rightarrow L$

$ID \rightarrow L(L|D)^*$

در مثال فوق ، قبل از ارائهٔ گرامر ، خلاصه سازی انجام دادیم که به این روش ، تعاریف منظم گفته می‌شود.

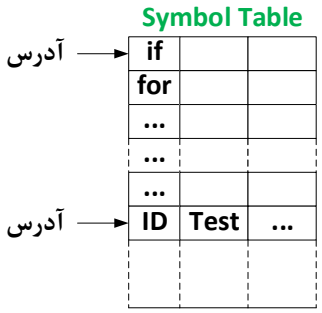
اگر بخواهیم برای مثال فوق ، یک NFA ارائه کنیم ، خواهیم داشت :



فرض کنید دستور زیر به این NFA ارسال می‌شود :

$Test := A * B$

نکات :



- Others شامل هر فضای خالی و یا هر نمادی است
- وقتی به وضعیت 2 می‌رسیم ، عبارت خوانده شده ، "$Test$" است. با علامت "*" که در بالای وضعیت 2 آمده ، یک نماد به عقب بر می‌گردیم. در اینصورت خواهیم داشت : "$Test$ "
- در صورتی که Token مورد نظر ، در جدول نمادها نباشد ، آنرا به جدول نمادها اضافه کرده و آدرس آن برگشت داده می‌شود. در غیر اینصورت (یعنی Token در جدول موجود است) ، تنها آدرس آن برگشت داده خواهد شد. این عمل که در پارامتر دوم از دستور زیر قرار دارد ، توسط تابعی به نام Install_ID() انجام می‌شود.

(آدرس Token شناسایی شده که در جدول Symbol Table قرار می‌گیرد ,ID) Return

الگوی تشخیص Keyword :

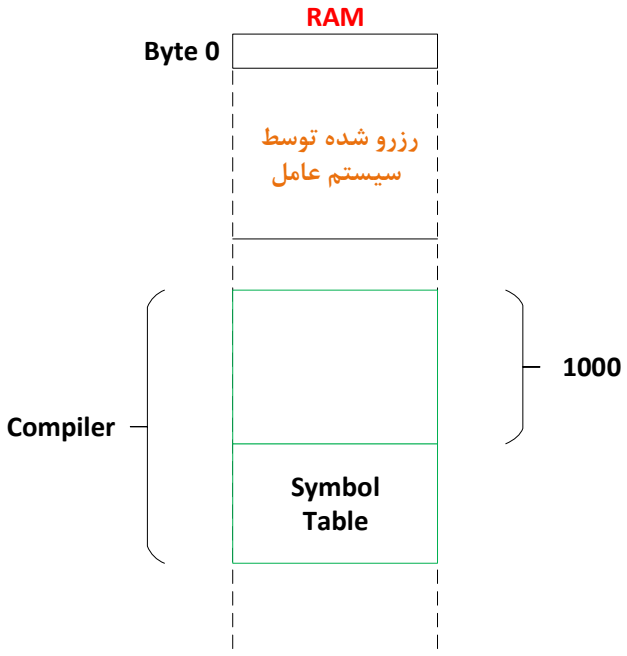
این مرحله هم دقیقاً همانند مرحلهٔ قبل است و همان الگوها را دارد. تنها تفاوت این است که در ابتدای زمان کامپایل ، کلیهٔ دستورات برنامه ، بصورت دستی در Symbol Table قرار داده می‌شود. به همین جهت است که به آنها Keyword گفته می‌شود (چون آنها ذخیره و رزرو شده‌اند)

نکات :

- اگر الگوی شناسایی شده ، جزء کلمات کلیدی باشد ، خروجی تابع GetToken در دستور زیر ، Keyword و در غیر اینصورت ID است.

(آدرس Token شناسایی شده که در جدول Symbol Table قرار می‌گیرد ,GetToken()) Return

- زمانیکه کامپایلر در حافظهٔ RAM قرار می‌گیرد ، جدول نمادها همواره از یک اختلاف آدرس ثابتی از ابتدای آدرس کامپایلر در حافظه قرار می‌گیرد. برای مثال اگر این اختلاف آدرس برابر هزار باشد ؛ داریم :



الگوی تشخیص NUM :

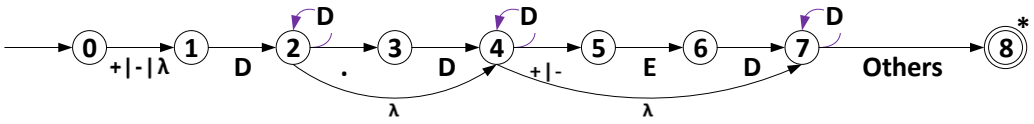
برای ساخت الگو ، در این مورد نیز سعی می‌کنیم برای تمامی حالات ممکن مثالی بزنیم :

$$9 \quad , \quad 912 \quad , \quad +9 \quad , \quad -9 \quad , \quad 9.5 \quad , \quad +9.52+E5 \quad , \quad -9.52+E5 \quad , \quad +9.52-E5 \quad , \quad -9.52-E5$$

که عبارت منظم آن بصورت زیر خواهد بود :

$$(+|-|\lambda).D.D^*.(\lambda|\bullet.D.D^*).(\lambda|(+|-).E.D.D^*) \quad \Rightarrow \quad .=Concat \quad , \quad \bullet=decimal\ \ point$$

NFA آن نیز بصورت زیر می‌باشد :



نکات :

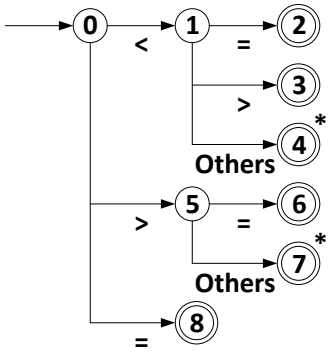
- Others شامل هر فضای خالی و یا هر نمادی به غیر از D است
- برای مثال ، NFA ارائه شده در فوق از عبارت A=573+E10+75 ، عدد 573+E10 را استخراج خواهد کرد.
- در صورتی که Token مورد نظر ، در جدول نمادها نباشد ، آنرا به جدول نمادها اضافه کرده و آدرس آن برگشت داده می‌شود. در غیر اینصورت (یعنی Token در جدول موجود است) ، تنها آدرس آن برگشت داده خواهد شد. این عمل که در پارامتر دوم از دستور زیر قرار دارد ، توسط تابعی به نام Install_NUM() انجام می‌شود.

Return (NUM, Install_NUM()) = Return (NUM, آدرس Token شناسایی شده که در جدول Symbol Table قرار می‌گیرد)

الگوی تشخیص عملگرهای مقایسه‌ای :

عملگرهای مقایسه‌ای در جدول نمادها قرار نمی‌گیرند و عبارتند از : $<>$, $=$, $\leq$, $<$, $\geq$, $>$. برای هر کدام از عملگرها ، یک عبارت منظم وجود دارد. عبارات منظم و NFA آن بصورت زیر است :

$$OP \rightarrow \ >|\geq|<|\leq|=|<>$$



Return (OP, $\leq$)

Return (OP, $<>$)

Return (OP, $<$)

Return (OP, $\geq$)

Return (OP, $>$)

Return (OP, $=$)

الگوی تشخیص رشته‌ها :

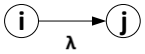
هر رشته‌ای ، در بین دبل کوتیشن (") قرار می‌گیرد. رشته‌ها نیز در جدول نمادها قرار نگرفته و برای آن تنها می‌توانیم عبارت منظم ارائه کنیم که عبارت است از :

$$".(L|D|...)^*."$$

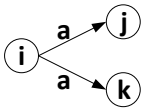
تفاوت‌های DFA و NFA :

سه تفاوت بین DFA و NFA عبارت است از :

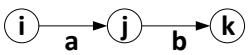
۱. در NFA می‌توانستیم با λ تغییر وضعیت دهیم که در DFA امکان پذیر نیست.



۲. در DFA به ازای هر نماد ، تنها به یک وضعیت می‌توانستیم برویم که در NFA اینطور نیست.



۳. اگر $\Sigma=\{a,b\}$ باشد ، در DFA به ازای هر الفبا ، یک خروجی باید از وضعیت وجود داشته باشد در حالیکه در NFA اینطور نیست.

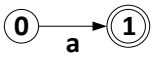


تبدیل عبارات منظم به NFA :

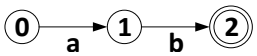
تا اینجا برای شناسایی توکن‌ها ، یک الگو یا عبارت منظم ارائه می‌کردیم. حال می‌خواهیم آنرا به NFA تبدیل کنیم که برای آن یک الگوریتم وجود دارد. توجه داشته باشید که تبدیل یک عبارت به NFA بسیار ساده‌تر از تبدیل آن به DFA است. پس ابتدا آنرا به NFA تبدیل کرده و سپس NFA را به DFA تبدیل می‌کنیم.

در زیر عبارت‌ها منظم کلی به همراه NFA آن مشخص شده :

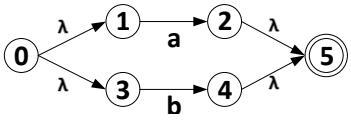
a :



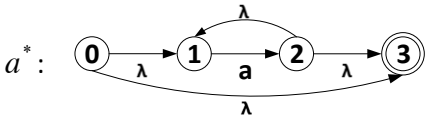
$a\ .\ b$:



$a\ |\ b$:



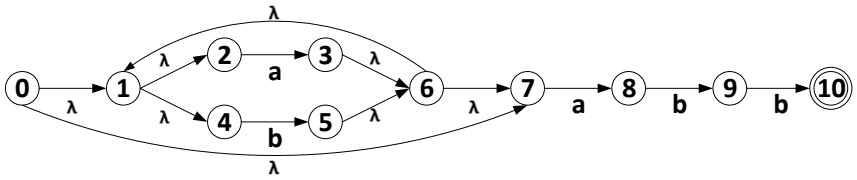
a^* :



واضح است که تمامی FAهای ارائه شده در فوق ، NFA هستند.

مثال : NFA معادل عبارت $(a|b)^*abb$ را رسم کنید.

برای اینگونه مثال‌ها ، ابتدا به سراغ داخلی‌ترین پرانتز می‌رویم. پس خواهیم داشت :



تعریف تابع $\lambda-Closure$:

$\lambda-Closure$ وضعیت a ، مجموعه‌ای از خود وضعیت a و همچنین وضعیت‌هایی است که از وضعیت a با λ می‌توان به آنها رفت. برای مثال ، $\lambda-Closure(0)$ در مثال بالا بصورت زیر خواهد بود :

$$\lambda-Closure(0)=\{0,1,2,4,7\}$$

تبدیل NFA به DFA :

این الگوریتم شامل دو مرحله است :

۱. بدست آوردن $\lambda-Closure$ برای هر وضعیت. اگر این مجموعه با مجموعه‌های قبلی بدست آمده متفاوت بود ، آنرا به عنوان یک وضعیت جدید در نظر می‌گیریم.

۲. بررسی کردن اینکه هر وضعیت از مجموعهٔ بدست آمده در مرحلهٔ قبل به ازای هر حرف از الفبا به کدام وضعیت می‌رود.

۳. در پایان ، وضعیت پایانی در هر کدام از مجموعه‌ها وجود داشت ، آن وضعیت جدید ، یک وضعیت پایانی خواهد بود.

مثال : DFA معادل در NFA بدست آمده در مثال قبلی را رسم کنید.

$$\lambda-Closure(0)=\{0,1,2,4,7\}=A$$

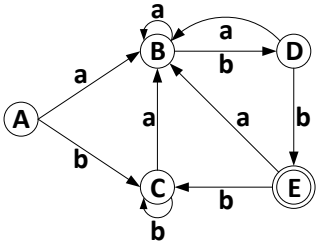
$$A\Rightarrow\begin{cases}a:\to\{3,8\} & \to\ \lambda-Closure(\{3,8\})=\{3,8,6,7,1,2,4\}=B\\ b:\to\{5\} & \to\ \lambda-Closure(\{5\})=\{5,6,7,1,2,4\}=C\end{cases}$$

$$B\Rightarrow\begin{cases}a:\to\{3,8\} & \to\ B\\ b:\to\{5,9\} & \to\ \lambda-Closure(\{5,9\})=\{5,9,6,7,1,2,4\}=D\end{cases}$$

$$C\Rightarrow\begin{cases}a:\to\{3,8\} & \to\ B\\ b:\to\{5\} & \to\ C\end{cases}$$

$$D\Rightarrow\begin{cases}a:\to\{3,8\} & \to\ B\\ b:\to\{5,10\} & \to\ \lambda-Closure(\{5,10\})=\{5,10,6,7,1,2,4\}=E\end{cases}$$

$$E\Rightarrow\begin{cases}a:\to\{3,8\} & \to\ B\\ b:\to\{5\} & \to\ C\end{cases}$$

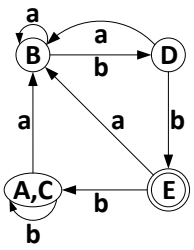


چون وضعیت شماره 10 تنها در مجموعهٔ E قرار داشت ، تنها وضعیت E یک وضعیت پایانی خواهد بود.

بهینه سازی DFA :

در این مرحله ، بایستی نودهایی که ادغام پذیر هستند را شناسایی کرده و با هم ادغام کنیم. در اولین گام ، می‌توان وضعیت‌های پایانی و غیر پایانی را در گروه‌های جداگانه قرار داد. سپس مشخص می‌کنیم که هر وضعیت با هر نماد از الفبا به کدام وضعیت درکدام گروه خواهد رفت. وضعیت‌هایی که رفتارهای یکسانی از خود نشان می‌دهند در یک گروه قرار می‌گیرند. این عملیات را تا جایی که وضعیت‌ها از هم جدا می‌شوند ، ادامه می‌دهیم.

مثال : DFA بهینه ، معادل DFA بدست آمده در مثال قبلی را رسم کنید.



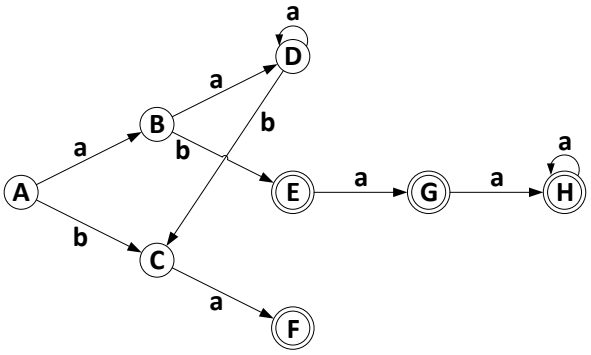
$$1:(A,B,C,D)\ ,\ 2:(E)\ \Rightarrow\ A\begin{cases}a:\to 1\\ b:\to 1\end{cases},\ B\begin{cases}a:\to 1\\ b:\to 1\end{cases},\ C\begin{cases}a:\to 1\\ b:\to 1\end{cases},\ D\begin{cases}a:\to 1\\ b:\to 2\end{cases}$$

$$1:(A,B,C)\ ,\ 2:(D)\ ,\ 3:(E)\ \Rightarrow\ A\begin{cases}a:\to 1\\ b:\to 1\end{cases},\ B\begin{cases}a:\to 1\\ b:\to 2\end{cases},\ C\begin{cases}a:\to 1\\ b:\to 1\end{cases}$$

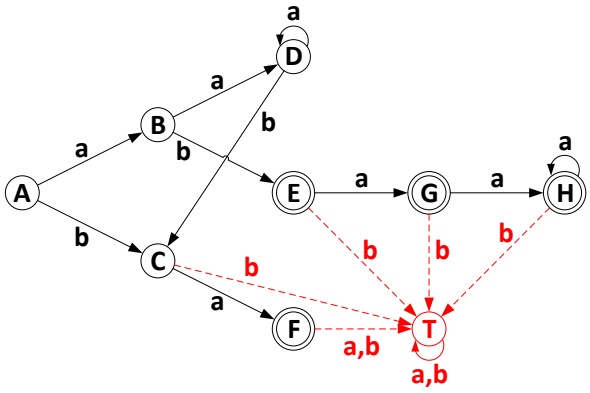
$$1:(A,C)\ ,\ 2:(B)\ ,\ 3:(D)\ ,\ 4:(E)\ \Rightarrow\ A\begin{cases}a:\to 2\\ b:\to 1\end{cases},\ C\begin{cases}a:\to 2\\ b:\to 1\end{cases}$$

وضعیت‌های A,C ادغام پذیر هستند.

مثال : NFA زیر (شکل یک) را به DFA بهینه تبدیل نمائید.



شکل یک



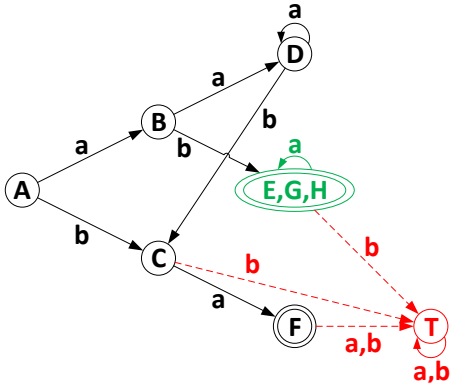
شکل دو

اگر بخواهیم NFA فوق را از روش $\lambda-Closure$ به DFA تبدیل کنیم ، بسیار زمان‌بر خواهد بود. اگر بتوانیم سه عاملی که تفاوت‌های بین NFA و DFA را بیان می‌کرد ، در NFA فوق بر طرف کنیم ، زودتر به DFA خواهیم رسید و سپس آنرا بهینه خواهیم کرد.

نکته : اگر تنها دلیل اینکه یک NFA ، DFA نیست این باشد که به ازای برخی از نماد الفبا ، خروجی از وضعیتی مشخص نباشد ، یعنی خروجی‌ها در بعضی وضعیت‌ها موجود نباشند ، می‌توان یک وضعیت جدید به NFA اضافه کرد و از تمام وضعیت‌های فوق ، با خروجی‌های ناقص به این وضعیت جدید رفت. به این وضعیت جدید اضافه شده ، وضعیت تله یا مرده گفته می‌شود. در صورت رفع تمام این خروجی‌های ناقص ، یک DFA در اختیار خواهیم داشت که می‌توان آنرا بهینه کرد (شکل دو).

$$\left.\begin{matrix}1:(A,B,C,D,T)\\ 2:(E,F,G,H)\end{matrix}\right\}\Rightarrow\begin{cases}A\begin{cases}a:\to 1\\ b:\to 1\end{cases},\ B\begin{cases}a:\to 1\\ b:\to 2\end{cases},\ C\begin{cases}a:\to 2\\ b:\to 1\end{cases},\ D\begin{cases}a:\to 1\\ b:\to 1\end{cases}\\ E\begin{cases}a:\to 2\\ b:\to 1\end{cases},\ F\begin{cases}a:\to 1\\ b:\to 1\end{cases},\ G\begin{cases}a:\to 2\\ b:\to 1\end{cases},\ H\begin{cases}a:\to 2\\ b:\to 1\end{cases},\ T\begin{cases}a:\to 1\\ b:\to 1\end{cases}\end{cases}$$

$$\left.\begin{matrix}1:(A,D,T)\ ,\ 5:(E,G,H)\\ 2:(B)\ ,\ 3:(C)\ ,\ 4:(F)\end{matrix}\right\}\Rightarrow\begin{cases}A\begin{cases}a:\to 2\\ b:\to 3\end{cases},\ D\begin{cases}a:\to 1\\ b:\to 3\end{cases},\ T\begin{cases}a:\to 1\\ b:\to 1\end{cases},\ E\begin{cases}a:\to 5\\ b:\to 1\end{cases},\ G\begin{cases}a:\to 5\\ b:\to 1\end{cases},\ H\begin{cases}a:\to 5\\ b:\to 1\end{cases}\end{cases}$$



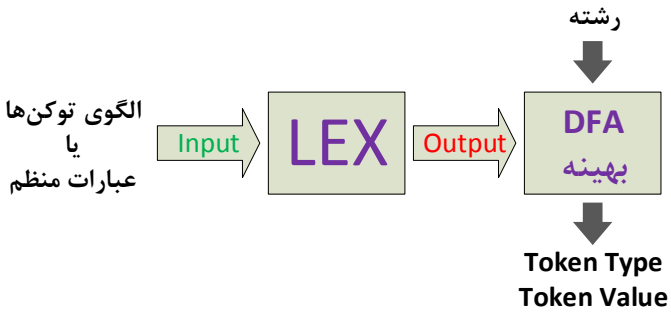
روش‌های ایجاد یا پیاده سازی تحلیل‌گر لغوی :

برای پیاده سازی این تحلیل‌گر ، سه روش وجود دارد که عبارت است از :

۱. با استفاده از زبان‌های سطح بالا (مثل C) : که ورودی آن یک رشته و خروجی آن Token Type و Token Value است. ولی عیب این روش آن است که زبان جدید به خصوصیات زبان پیاده سازی وابسته خواهد بود (مثل Java که به C وابسته است)

۲. با استفاده از زبان اسمبلی : زبان اسمبلی با زبان ماشین نوشته شده و معنی آن این است که دستورات آن توسط سخت افزار قابل شناسایی است. عیب این روش ، سخت بودن زبان اسمبلی است.

۳. با استفاده از ابزار LEX : که بصورت رایگان است. ورودی این ابزار ، الگوی توکن‌ها (که بصورت عبارات منظم است) و خروجی آن ، DFA بهینه است.



ساختار زبان LEX :

هر برنامه شامل سه بخش اصلی است که هر بخش با علامت "%%" از بخش دیگر جدا می‌شود. این سه بخش عبارتند از :

۱. بخش اعلان‌ها یا Declarations
۲. بخش قوانین
۳. بخش توابع کمکی

مثال : تحلیل‌گر لغوی را برای شناسایی ID و NUM پیاده سازی نمائید.

$$\left.\begin{aligned} D &\to [0\cdots 9] \\ L &\to [a\cdots z],[A\cdots Z] \\ ID &\to L.(L\mid D)^* \\ NUM &\to \cdots \end{aligned}\right\}$$ گوی ت وک ن ها

%%

$ID \to return\{getToken(), install_ID()\}$

…

%%

$getToken()$

{

ت عریف ب دنۀ ت ابع ک م ک ی

ب ه همان س ینۀ ت کس زب ان س ی

}

…

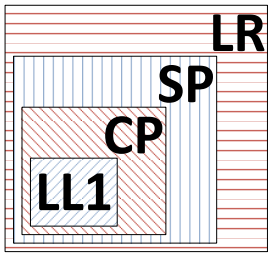
« فصل سوم : فاز تحلیل‌گر نحوی »

تشخیص ترتیب توکن‌ها (اینکه برای دستور شرطی if ، اول if باشد ، بعد then و سپس else) در این فصل انجام خواهد شد. در اینگونه دستورات ، معمولاً از پرانتز استفاده می‌شود و چون تعداد پرانتزهای باز و بسته بایستی برابر باشد ، تعداد نوع پرانتزها به یکدیگر وابسته بوده و مشخص است که باید از زبان‌های مستقل از متن استفاده شود. این نوع زبان ، تنها با گرامر بیان شده و سپس برای آنها یک PDA ارائه خواهیم کرد. ورودی این PDA ، دنباله‌ای از توکن‌ها و خروجی آن ، درخت تجزیه است که مهمترین وظیفهٔ PDA است. ایجاد درخت تجزیه ، به چندین شکل انجام می‌شود که عبارت است از :

۱. روش‌های بالا به پائین : که تنها گرامرهای نوع LL1 را با کارایی بالا پشتیبانی می‌کنند
۲. روش‌های پائین به بالا : که عبارتند از :
   a. CP
   b. SP
   c. LR : که خود عبارت است از :
      i. SLR
      ii. CLR
      iii. LALR

که هر کدام یکسری از گرامرها را با کارایی بالا پشتیبانی می‌کنند

گرامرهای مستقل از متن



دلایل استفاده از گرامرهای مستقل از متن :

به دو دلیل در این تحلیل‌گر ، از گرامر مستقل از متن استفاده می‌کنیم :

۱. زبان مستقل از متن ، ابزاری قوی‌تر از زبان‌های منظم می‌باشد. زبان‌های منظم ، قادر به بیان ساختارهای تو در تو نمی‌باشد.
۲. استفاده از یک ابزار جدید ، ساختار مستقل طراحی کامپایلرها را واضح‌تر بیان می‌کند.

گرامر این نوع زبان ، بصورت $A \to \alpha$ است ؛ یعنی در سمت چپ ، تنها یک متغیر و در سمت راست ، هر چیزی

نمایش ساختار دستورات با گرامر مستقل از متن :

فرض کنید می‌خواهیم ساختار دستور if را با این گرامر نشان دهیم. این ساختار به دو صورت قابل نمایش است.

حالت اول ، نمایش به فرم BNF است. در این فرم ، متغیرها یا پایانی‌ها با علامت کوچکتر بزرگتر ، از سایرین جدا می‌شود. پس خواهیم داشت :

$$< if\ -stmt > \ := \ if\ < \text{expr} > then < stmt > \ | \ if\ < \text{expr} > then < stmt > else < stmt >$$

در حالت دوم خواهیم داشت :

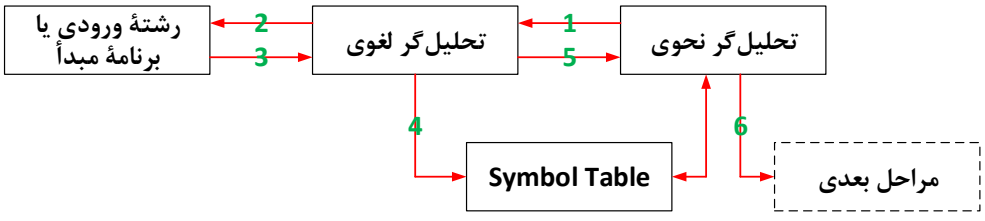
$$S \to iEtS \mid iEtSeS$$
$$E \to b$$

همانطور که مشخص است این گرامر ، ترتیب توکن‌ها را نمایش می‌دهد. در تحلیل نحوی ، منظور از هر پایانی یا نماد گرامر یا نماد الفبای گرامر ، یک توکنی می‌باشد که توسط تحلیل‌گر لغوی شناسایی شده است.

نقش تحلیل‌گر نحوی در کامپایلر :

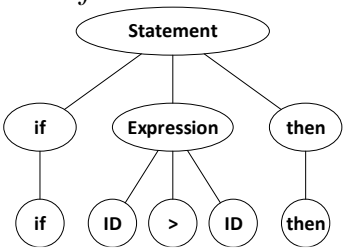
ترتیب اجرا شدن فازها در کامپایلر بصورت زیر است :

۱. عملیات از تحلیل‌گر نحوی شروع شده و درخواست خواندن توکن بعدی را به تحلیل‌گر لغوی می‌دهد.
۲. تحلیل‌گر لغوی شروع به خواندن رشتهٔ ورودی می‌کند.
۳. رشته‌های خوانده شده به تحلیل‌گر لغوی برای پردازش ارسال می‌شوند.
۴. تحلیل‌گر لغوی ، توکن‌ها را از رشته شناسایی کرده و درون جدول نمادها قرار می‌دهد.
۵. تحلیل‌گر لغوی ، آدرس توکن‌های شناسایی شده را به تحلیل‌گر نحوی می‌دهد.
۶. تحلیل‌گر نحوی ، با استفاده از توکن‌ها و با توجه به گرامر مستقل از متنی که ترتیب توکن‌ها را نشان می‌دهد ، درخت تجزیه را برای استفاده در مراحل بعدی ایجاد می‌کند.



(مثال درخت تجزیه)

$if\ \ A > B\ \ then$



روش‌های ساخت درخت تجزیه :

به روش‌هایی که منجر به ساخت درخت تجزیه می‌شود ، روش‌های تجزیه یا Parse گفته می‌شود که عبارتند از :

۱. روش‌های استاندارد تجزیه ۲. روش‌های بالا به پائین ۳. روش‌های پائین به بالا

روش‌های استاندارد تجزیه :

این روش با استفاده از اشتقاق انجام می‌شود. اگر طول رشته را با w و تعداد قوانین را با p نشان دهیم ، پیچیدگی زمانی آنها بصورت $O\left(|P|\right)^{2|W|}$ است. روش‌هایی نیز به نام CYK هستند که پیچیدگی زمانی آنها بصورت $O\left(|W|^3\right)$ است. این روش‌ها کارایی لازم را ندارند ؛ زیرا بصورت سعی و خطا بوده و پیچیدگی‌های زمانی دارند. روش‌هایی مطلوب است که پیچیدگی زمانی آنها خطی باشد.

روش‌های بالا به پائین یا بازگشتی کاهشی (Recursive Descent) :

که تنها یک روش تجزیه از آن وجود دارد و به آن روش تجزیهٔ پیش‌نگر یا Predictive گفته می‌شود و در واقع همان اشتقاق است. پیچیدگی زمانی آن $O\left(|W|\right)$ است. این روش بر روی همهٔ گرامرهای مستقل از متن قابل اجرا نبوده و تنها روی گرامرهای LL1 قابل اعمال هستند.

روش‌های پائین به بالا یا Shift Reduce :

این روش ، اشتقاق برعکس است ؛ یعنی از برگ شروع شده و به ریشه ختم می‌شود. این روش شامل سه روش با پیچیدگی زمانی $O\left(|W|\right)$ است که عبارتند از :

۱. روش تجزیهٔ تقدم عملگر : قابل اعمال بر روی گرامرهای عمل گرا
۲. روش تجزیهٔ تقدم ساده : قابل اعمال بر روی گرامرهای عمل گرای توسعه یافته
۳. روش تجزیهٔ LR : قابل اعمال بر روی گرامرهای LR(1) است و تقریباً تمامی گرامرهای مستقل از متن را پوشش می‌دهد

گرامرهای تحت پوشش کمتر ↑	روش تجزیهٔ پیش‌نگر	روش‌های ساده‌تر ↑
	روش تجزیهٔ تقدم عملگر	
	روش تجزیهٔ تقدم ساده	
گرامرهای تحت پوشش بیشتر ↓	روش تجزیهٔ LR	روش‌های سخت‌تر ↓

حال به تعریف یکسری مفاهیم ابتدائی می‌پردازیم.

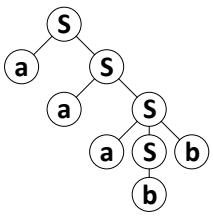
اشتقاق :

که همان روش بالا به پائین است و به توصیف بالا به پائین برای نمایش رشته‌های تولید شده از یک گرامر گفته می‌شود.

مثال :

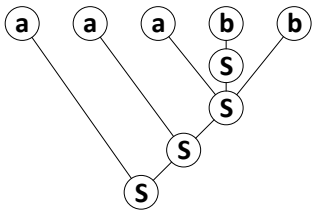
$S \to aSb \mid aS \mid b \quad , \quad W = aaabb$

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaSb \Rightarrow aaabb$



می‌توانیم بصورت عکس نیز عمل کنیم ؛ یعنی از برگ به ریشه برسیم ؛ یعنی پائین به بالا

$aaabb \Rightarrow aaaSb \Rightarrow aaS \Rightarrow aS \Rightarrow S$



اشتقاق Right Most و Left Most :

اشتقاق Left Most یعنی همواره سمت چپ‌ترین متغیر و اشتقاق Right Most یعنی همواره سمت راست‌ترین متغیر ، جایگزین می‌شوند. مثال :

$S \to AS \mid b$

$A \to BA \mid aA \mid a \quad , \quad W = bbab$

$B \to bB \mid b$

$L.M \quad : \quad S \Rightarrow AS \Rightarrow BAS \Rightarrow bBAS \Rightarrow bbAS \Rightarrow bbaS \Rightarrow bbab$

$R.M \quad : \quad A \Rightarrow AS \Rightarrow Ab \Rightarrow BAb \Rightarrow Bab \Rightarrow bBab \Rightarrow bbab$

توجه داشته باشید که ترکیبی از این دو اشتقاق وجود ندارد.

ابهام در گرامر :

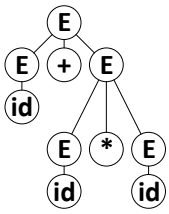
در صورتیکه برای یک گرامر ، بیش از یک اشتقاق L.M یا بیش از یک اشتقاق R.M بدست آید ، در آن گرامر ابهام وجود دارد.

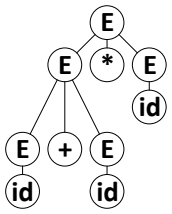
مثال : (id یک توکن است) سعی می‌کنیم برای گرامر زیر دو اشتقاق L.M بدست آوریم :

$$E \to E+E \mid E*E \mid (E) \mid id \quad , \quad W = id + id * id$$

$$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$$

$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$$



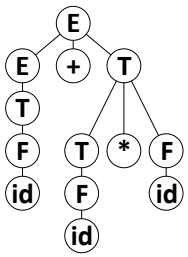


گرامر مبهم مطلوب نیست زیرا نتیجه‌های مختلفی دارد ؛ یعنی مفهوم مختلفی خواهد داشت. برای مثال اگر در مثال قبلی بخواهیم نتیجهٔ عبارت $W = 2+4*3$ را محاسبه کنیم ، با اشتقاق سمت چپ $W = 14$ و با اشتقاق سمت راست $W = 18$ خواهد شد.

دلیل وجود ابهام ، در نظر نگرفتن اولویت عملگرها در گرامر است. در اینصورت گرامر تنها ترتیب را نمایش می‌دهد. پس بایستی در گرامرهای مبهم ، رفع ابهام شود ولی چون قاعدهٔ کلی برای آن وجود ندارد ، کار آسانی نیست. بایستی اول بدانیم که چه عاملی باعث ابهام شده است. در کل بایستی گرامر را بصورتی بنویسیم که اولویت‌ها در نظر گرفته شوند که برای این امر ، معمولاً از چند متغیر کمکی استفاده می‌کنند. مثال :

$$\Leftarrow \begin{array}{l} E \to E+T \mid T \\ T \to T*F \mid F \\ F \to (E) \mid id \end{array}$$

در اینصورت اولویت از بالا به پایین زیاد می‌شود.
درخت رشتهٔ مثال قبلی نیز با این گرامر ، تنها یک درخت خواهد بود که کمی طولانی تر هم شده ولی ابهامی ندارد



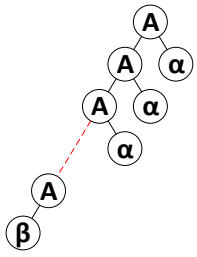
چپ گردی یا Left Recursive :

یعنی اینکه در یک گرامر ، متغیر سمت چپ ، بالافاصله در سمت راست نیز تکرار شود. صورت کلی آن بصورت $A \to A\alpha \mid \beta$ است و وجود آن ، باعث ایجاد حلقهٔ بی‌پایان در گرامر می‌شود ؛ به همین علت خوب نیست و باید حذف شود. مثال :

$$\underline{\underline{E}} \to \underline{\underline{E}} + T$$
$$\underline{\underline{T}} \to \underline{\underline{T}} * F \mid F$$

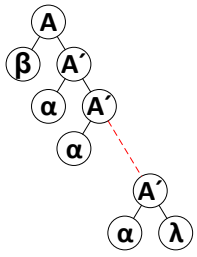
قانون کلی حذف آن بصورت زیر است :

$$A \to A\alpha \mid \beta \quad \Rightarrow$$



رفع چپ گردی

$$\left.\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \lambda \end{array}\right\} \Rightarrow$$



مثال : می‌خواهیم چپ گردی در گرامر ریاضی بدست آمدهٔ مثال قبلی را حذف کنیم :

$$\left.\begin{array}{l} E \to E+T \mid T \\ T \to T*F \mid F \\ F \to (E) \mid id \end{array}\right\} \Rightarrow \left\{\begin{array}{lll} \alpha = +T & , & \beta = T \\ \alpha = *F & , & \beta = F \\ -- & & \end{array}\right. \Rightarrow\Rightarrow\Rightarrow \begin{array}{l} E \to TE' \\ E' \to +TE' \mid \lambda \\ T \to FT' \\ T' \to *FT' \mid \lambda \\ F \to (E) \mid id \end{array}$$

فاکتور چپ :

که دقیقاً مانند فاکتورگیری در ریاضی است. در این حالت ، یک عامل مشترکی در قوانین وجود دارد که بایستی حذف شود. صورت کلی آن بصورت زیر است :

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \gamma \quad \Rightarrow \text{ که بایستی به این فرم تبدیل شود } \Rightarrow \quad \begin{matrix} A \to \alpha B \mid \gamma \\ B \to \beta_1 \mid \beta_2 \mid ... \end{matrix}$$

فاکتور چپ باعث خواهد شد که در اشتقاق ، تا جایی که مشترک است به درستی خوانده شده و سپس انتخاب قانون برای درخت تجزیه به سختی صورت گیرد. پس فاکتور چپ نیز مطلوب نبوده و بایستی حذف شود.

مثال : حذف فاکتور چپ از گرامر دستور if

$$\left.\begin{matrix} S \to iEtS \mid iEtSeS \mid a \\ E \to b \end{matrix}\right\} \Rightarrow \{\alpha = iEtS \quad , \quad \beta_1 = \lambda \quad , \quad \beta_2 = eS \quad \Rightarrow\Rightarrow\Rightarrow \quad \begin{matrix} S \to iEtSS' \mid a \\ S' \to \lambda \mid eS \\ E \to b \end{matrix}$$

مروری بر تحلیل‌گر نحوی :

وظیفهٔ این تحلیل‌گر ، شناسایی ترتیب توکن‌هاست که با استفاده از زبان‌های مستقل از متن صورت می‌گیرد. این زبان بوسیلهٔ گرامر مشتقل از متن قابل نمایش است.
از طرفی زبان‌های مستقل از متن دارای یکسری مشکلات به شرح زیر است :

۱. ابهام : که باعث اثر گذاری بر روی نتایج خواهد شد
۲. چپ گردی : که اثر آن بر روی پیاده سازی است
۳. فاکتور چپ : که باعث پیچیده شده انتخاب قانون مناسب برای تجزیه خواهد شد

گرامر مستقل از متنی که سه عمل فوق را روی آن انجام دهیم ، گرامر LL(1) گفته می‌شود که تنها گرامری است که تجزیهٔ بالا به پائین روی آن انجام می‌شود.
گرامرهای مستقل از متنی که سه عمل فوق را نتوان روی آنها انجام داد ، گرامرهای ذاتاً مبهم می‌گویند که تجزیهٔ پائین به بالا روی آنها انجام خواهد شد.

نحوهٔ عملکرد تجزیهٔ بالا به پائین یا اشتقاق :

فرض کنید می‌خواهیم اشتقاق رشتهٔ زیر را با گرامر داده شده بدست آوریم :

$$\begin{matrix} S \to cAd \\ A \to ab \mid a \end{matrix} \quad , \quad W = cad$$

تحلیل‌گر نحوی ، اولین حرف رشته را می‌خواند (c). سپس در قوانین ، به جستجوی قوانینی می‌پردازد که با حرف خوانده شده مطابقت داشته باشند ؛ که در اینجا تنها یک قانون یافت می‌شود :

$$S \Rightarrow cAd$$

مرحلهٔ قبل را برای حرف دوم نیز تکرار می‌کند :

$$S \Rightarrow cAd \Rightarrow cabd$$

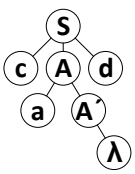
در اینجا ، قانون‌ها تمام می‌شود ولی به جواب صحیح نرسیدیم. در اینجا عمل Back Tracking صورت گرفته و یک قانون دیگری انتخاب می‌شود :

$$S \Rightarrow cAd \Rightarrow cad$$

گرامری که در آن منجر به برگشت به عقب شویم مطلوب نیست زیرا این نوع گرامرها دارای پیچیدگی‌های زمانی هستند ($O\left(|P|\right)^{2|W|} = O(2^6)$) که اصلاً برای تجزیه مطلوب نیستند.
ولی اگر دقت کنیم خواهیم فهمید که گرامر فوق ، گرامر LL(1) نیست. در این گرامر ، ابهام و چپ گردی وجود ندارد ؛ ولی حاوی فاکتور چپ است و لذا باید آنرا حذف کرد. در اینصورت خواهیم داشت :

$$\begin{matrix} S \to cAd \\ A \to aA' \\ A' \to b \mid \lambda \end{matrix} \quad , \quad W = cad$$

در اینصورت ، اشتقاق آن نیز بصورت زیر خواهد بود :



$$S \Rightarrow cAd \Rightarrow caA'd \Rightarrow ca\lambda d \Rightarrow cad$$

تجزیه کننده‌های پیش‌نگر :

به تجزیه کننده‌های بالا به پائینی که تجزیه را به کمک گرامرهای LL(1) انجام داده و در آنها برگشت به عقب یا BackTracking وجود ندارد ، تجزیه کننده‌های پیش‌نگر گفته می‌شود.

$$L \Rightarrow Left \Rightarrow$$ خواندن نمادهای ورودی از چپ به راست

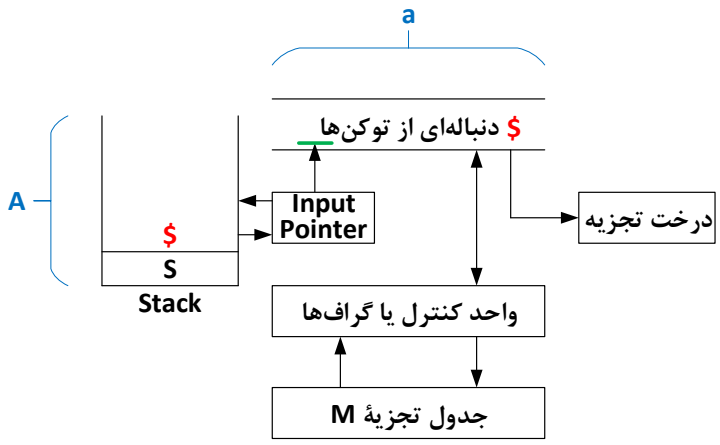
$$L \Rightarrow Left\ Most \Rightarrow$$ ایجاد درخت تجزیه به روش اول چپ

$$1 \Rightarrow One\ Character \Rightarrow$$ در هر مرحله ، تنها یک نماد خوانده می‌شود

گرامر LL(k) ، گرامری است که قانون درست برای تجزیه را با خواندن k نماد ورودی می‌تواند تشخیص دهد. البته کامپایلرها تنها از گرامرهای LL(1) استفاده می‌کنند. چرا به آنها پیش‌نگر گرفته می‌شود ؟ زیرا هر بار یک نماد خوانده شده و با توجه به آن قانون را انتخاب می‌کند.

پیاده سازی تجزیه کنندهٔ پیش‌نگر :

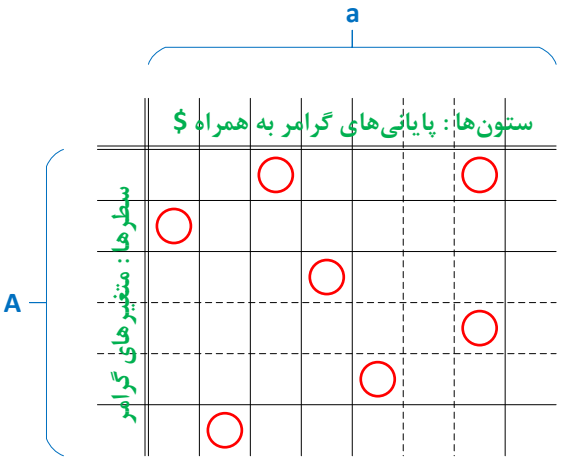
پیاده سازی تجزیه کننده ، توسط PDA انجام خواهد شد. قسمت‌های یک PDA در شکل زیر نشان داده شده است :



- A : نمادهای بالای Stack یا متغیرها
- a : نمادهای ورودی یا پایانی‌ها
- S در ابتدای Stack ، نماد شروع گرامر است.
- ابتدا و انتهای رشته را ، $ مشخص می‌کند که ابتدای آنرا بصورت دستی وارد Stack می‌کنیم. به همین علت است که تجزیه از بالا یا ریشه شروع می‌شود. در انتها ، هم داخل Stack به $ رسیده‌ایم و هم کاراکتر ورودی $ خواهد بود.

ساختار جدول تجزیهٔ M :

این جدول باعث خواهد شد که قانون درستی برای تجزیه انتخاب شود. ساختار این جدول ، در شکل زیر نشان داده شده است :



- ساختار جدول تجزیه ، یک آرایهٔ دو بُعدی است.
- جدول تجزیهٔ M : با توجه به نماد ورودی جاری و بالای Stack ، قانون درست برای تجزیه را انتخاب می‌کند.
- M[A,a] نشاندهندهٔ یک خانه است که دو حالت دارد :
  - قانونی که برای اشتقاق بایستی استفاده شود.
  - در صورت خالی بودن خانه ، رشته درست نبوده و پذیرفته نخواهد شد.
- در این جدول ، هر خانه حداکثر یک قانون دارد. در غیر اینصورت به این معنی است که گرامر آن LL(1) نیست (مبهم خواهد بود)
- a : ورودی جاری یا Symbol Table

در تجزیه به روش پیش‌نگر ، با دو عامل مهم برخورد خواهیم کرد که عبارتند از :

۱. عملکرد تجزیه کننده که طی یک الگوریتم و با یک مثال نشان داده خواهد شد.
۲. ساختن جدول تجزیهٔ M

عملکرد تجزیه کنندهٔ پیش‌نگر :

مثال : با توجه به گرامر و جدول تجزیهٔ داده شده ، رشتهٔ داده شده در زیر را تجزیه نمائید :

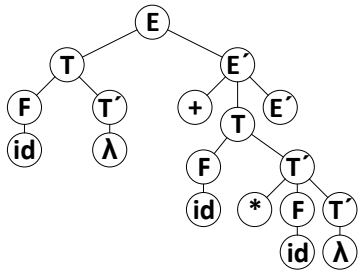
1. $E \to TE'$
2. $E' \to +TE'$
3. $E' \to \lambda$
4. $T \to FT'$
5. $T' \to *FT'$
6. $T' \to \lambda$
7. $F \to id$
8. $F \to (E)$

$$W = id + id * id$$

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

Stack	Input	Action
$E	$id + id * id$	$E \to TE'$
$ET	$id + id * id$	$T \to FT'$
$ETF	$id + id * id$	$F \to id$
$ETid	$id + id * id$	delete id
$ET'	$+id * id$	$T' \to \lambda$
$E'	$+id * id$	$E' \to +TE'$
$ET+	$+id * id$	delete +
$ET	$id * id$	$T \to FT'$
$ETF	$id * id$	$F \to id$
$ETid	$id * id$	delete id
$ET'	$*id$	$T' \to *FT'$
$ETF*	$*id$	delete *
$ETF	id	$F \to id$
$ETid	id	delete id
$ET'	$	$T' \to \lambda$
$E'	$	$E' \to \lambda$
$	$	Accept

همزمان می‌توانیم درخت آنرا نیز رسم کنیم : در اینصورت ، از نمادهای بالای Stack و یا سطرهای خوانده شده از جدول تجزیه ، برای هر گره استفاده خواهیم کرد :



- جدولی رسم می‌کنیم که در آن ، هم محتوای Stack ، هم ورودی و هم عملیاتی که بایستی انجام شود ، نشان دهد.
- با توجه به نماد شروع رشته و همچنین نمادی که در بالای Stack قرار دارد ، قانون مناسبی را انتخاب می‌کنیم.
- سمت چپ قانون انتخابی را حذف و به جای آخرین نماد موجود در Stack ، سمت راست قانون انتخاب شده را بصورت برعکس به جای آن قرار می‌دهیم.
- دو مرحلهٔ قبلی را تا جایی ادامه می‌دهیم که نماد بالای Stack و نماد خوانده شده از ورودی برابر شوند.
- نماد فوق را هم از بالای Stack و هم از ورودی جاری حذف کرده و دوباره الگوریتم فوق را تکرار می‌کنیم.
- در پایان اگر نماد ورودی و نماد بالای Stack ، هر دو $ شد ، رشته پذیرفته خواهد شد.

کل الگوریتم فوق را می‌توان در سه قانون خلاصه کرد :

۱. اگر $A = a = \$$ شود ، آنگاه رشته پذیرفته خواهد شد.
۲. اگر $A = a \neq \$$ شود ، آنگاه بایستی A, a را حذف کنیم.
۳. اگر $A \neq a$ شود ، آنگاه خانهٔ $M[A, a]$ بررسی خواهد شد.

روش ساختن جدول تجزیهٔ M :

این جدول تجزیه به کمک دو تابع به شرح زیر ساخته می‌شود که ابتدا بایستی بررسی کنیم که این توابع چه عملی انجام می‌دهند :

۱. First() ۲. Fallow()

تابع First() :

ورودی این تابع ، یک شکل جمله‌ای است ؛ یعنی ترکیبی از متغیرها و پایانی‌ها که سه حالت برای ورودی آن رخ خواهد داد :

۱. ورودی تنها یک پایانی باشد. مثال :

$$First(+) = +$$

۲. ورودی ، شکل جمله‌ای که با یک پایانی شروع می‌شود ، باشد. مثال :

$$First(+TE') = +$$

۳. ورودی یک متغیر باشد ؛ یا شکل جمله‌ای که با یک متغیر شروع می‌شود ، باشد. مثال :

$$First(FT') = \{(, id\}$$

$$\begin{aligned} &E \to TE' \\ &E' \to +TE' \mid \lambda \\ &T \to FT' \\ &T' \to *FT' \mid \lambda \\ &F \to (E) \mid id \end{aligned}$$

این تابع در واقع ، اولین پایانی است که متغیر در اشتقاق ، با آن می‌تواند شروع شود. در زیر نیز چندین مثال دیگر ذکر می‌کنیم :

$$First(T): \; T \Rightarrow FT' \Rightarrow idT' \quad , \quad T \Rightarrow FT' \Rightarrow (E)T' \quad \Rightarrow \quad First(T) = \{(, id\}$$

$$First(E') = \{+, \lambda\} \qquad First(T') = \{*, \lambda\} \qquad First(F) = First(E) = \{(, id\}$$

نکته :

در بعضی مراجع ، λ را جزء خروجی تابع First() نمی‌دانند و قبول نمی‌کنند و در برخی منابع آنرا قبول می‌کنند.

قضیه : اگر قوانینی بصورت $X \to Y_1 Y_2 Y_3 .. Y_n$ داشته باشیم و $Y_1 \Rightarrow \lambda$ ، آنگاه $First(Y_2)$ در $First(X)$ نیز وجود خواهد داشت. همچنین اگر $Y_1 \Rightarrow \lambda \quad , \quad Y_2 \Rightarrow \lambda$ آنگاه $First(Y_3)$ در $First(X)$ نیز وجود خواهد داشت. (و الی آخر)

مثال :

$$\begin{aligned} &S \to AB \\ &A \to aA \mid \lambda \\ &B \to bB \mid b \end{aligned} \qquad First(S): \; S \Rightarrow AB \Rightarrow aAB... \quad , \quad S \Rightarrow AB \Rightarrow B \Rightarrow bB... \quad \Rightarrow \quad First(S) = \{a, b\}$$

تابع Fallow() :

این تابع ، تنها برای متغیرها حساب می‌شود ؛ یعنی ورودی تابع ، تنها متغیر است. منظور از $Fallow(A)$ ، اولین پایانی است که بعد از A در آن گرامر دیده می‌شود که البته باید A را در قسمت راست قانون‌ها جستجو کرد.

قضیه :

$$X \to \alpha A \beta \quad \Rightarrow \quad \begin{cases} Fallow(A) = First(\beta) \\ if \; \beta = \lambda \; Or \; \beta \overset{*}{\Rightarrow} \lambda \quad \Rightarrow \quad Fallow(A) = Fallow(X) \end{cases}$$

نکته :

اگر A نماد شروع گرامر باشد ، آنگاه $\$ \in Fallow(A)$

مثال :

$$\begin{aligned} &E \to TE' \\ &E' \to +TE' \mid \lambda \\ &T \to FT' \\ &T' \to *FT' \mid \lambda \\ &F \to (E) \mid id \end{aligned}$$

$$Fallow(E) = Fallow(E') = \{), \$\}$$

$$Fallow(T) = Fallow(T') = \{First(E'), Fallow(E)\} = \{+,), \$\}$$

$$Fallow(F) = \{First(T'), Fallow(T)\} = \{*, +,), \$\}$$

مثال : برای گرامر زیر ، جدول تجزیهٔ M را ایجاد نمائید :

توجه داشته باشید که هدف ما در این قسمت ، بهینه سازی یک گرامر نیست و لذا اگر گرامر بهینه شده هم نباشد ، بایستی درخت تجزیه برای همین گرامر بهینه نشده رسم شود. پس به هیچ عنوان در گرامر تغییری نخواهیم داد.

1. $E \to TE'$
2. $E' \to +TE'$
3. $E' \to \lambda$
4. $T \to FT'$
5. $T' \to *FT'$
6. $T' \to \lambda$
7. $F \to id$
8. $F \to (E)$

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

روش کار به این صورت است :

- برای هر قانون موجود در گرامر ، $First()$ سمت چپ قانون را محاسبه می‌کنیم.
- در جدول ، به ازای آن متغیر و مجموعهٔ بدست آمده ، قانون مورد نظر را در آن قسمت درج می‌کنیم.

1 : $First(E) = First(T) = First(F) = \{id, (\} \Rightarrow M[E, id] = M[E, (] =$ 1

2 : $First(E') = \{+\} \Rightarrow M[E', +] =$ 2

3 : $First(E') = \lambda \Rightarrow First(E') = Fallow(E') = Fallow(E) = \{\$,)\} \Rightarrow M[E', \$] = M[E',)] =$ 3

4 : $First(T) = First(F) = \{id, (\} \Rightarrow M[T, id] = M[T, (] =$ 4

5 : $First(T') = \{*\} \Rightarrow M[T', *] =$ 5

6 : $First(T') = \lambda \Rightarrow First(T') = Fallow(T') = \{First(E'), Fallow(E)\} = \{+, \$,)\}$ 6

$\Rightarrow M[T', +] = M[T', \$] = M[T',)] =$

7 : $First(F) = First(id) = \{id\} \Rightarrow M[F, id] =$ 7

8 : $First(F) = First(() = \{(\} \Rightarrow M[F, (] =$ 8

الگوریتم کلی آن بصورت زیر است :

۱. اگر قانونی به شکل $A \to \alpha$ داشته باشیم ، آنگاه $M[A, a] = A \to \alpha$ که در آن $a \in First(\alpha)$

۲. اگر قانونی به شکل $A \to \lambda$ یا $\alpha \overset{*}{\Rightarrow} \lambda$, $A \to \alpha$ داشته باشیم ، آنگاه $M[A, a] = \begin{cases} A \to \lambda \\ A \to \alpha \end{cases}$ که در آن $a \in Fallow(\alpha)$

مثال : برای گرامر زیر ، جدول تجزیهٔ M را ایجاد نمائید :

$S \to iEtSS'$
$S' \to eS \mid \lambda$ $\Rightarrow$
$E \to b$

1. $S \to iEtSS'$
2. $S' \to eS$
3. $S' \to \lambda$
4. $E \to b$

A \ a	i	t	e	b	$
S	1				
S'			2 3		3
E				4	

1 : $First(S) = \{i\} \Rightarrow M[S, i] =$ 1

2 : $First(S') = \{e\} \Rightarrow M[S', e] =$ 2

3 : $First(S') = \lambda \Rightarrow First(S') = Fallow(S') = Fallow(S) = \{\$, First(S')\} = \{\$, e\}$ 3

$\Rightarrow M[S', \$] = M[S', e] =$

4 : $First(E) = \{b\} \Rightarrow M[E, b] =$ 4

تشخیص گرامرهای LL(1) بدون رسم جدول تجزیهٔ M :

گرامرهای LL(1) دارای سه شرط بوده و در صورتی که هر کدام را نداشته باشد ، گرامر LL(1) نخواهد بود. این شروط عبارتند از :

۱. اگر قانونی به شکل $A \to \alpha \mid \beta$ داشته باشیم ؛ یعنی یک متغیر چند قانون را تولید کند ، آنگاه $First(\alpha) \bigcap First(\beta) = \varnothing$ باشد ؛ یعنی اشتراک این دو مجموعه باید تهی باشد.

۲. اگر قانونی به شکل $A \to \alpha \mid \beta$ داشته باشیم ، حداکثر یکی از متغیرهای α یا β به λ ختم شود.

۳. اگر داشته باشیم : $\beta \overset{*}{\Rightarrow} \lambda$ ؛ آنگاه $Fallow(A) \bigcap First(\alpha) = \varnothing$ باشد ؛ یعنی اشتراک این دو مجموعه باید تهی باشد.

مثال : مشخص نمائید کدامیک از گرامرهای زیر ، LL(1) است ؟

$S \to Ab \mid Bb$
$A \to aB$
$B \to abCd$
$C \to \lambda$

$First(Ab) = \{a\}$
$First(Bb) = \{a\}$
چون عضو مشترک دارند
پس گرامر LL(1) نیست

$S \to aAb \mid aBb$
$A \to aB$
$B \to abCd$
$C \to \lambda$

داریم :
$M[S,a] =$ قانون اول
$M[S,a] =$ قانون دوم
مسلماً گرامر LL(1) نیست

$S \to aAb$
$A \to aB$
$B \to abCd$
$C \to \lambda$

برای گرامرهایی از این دست ، که هر حرف در بالای Stack ، تنها یک قانون را تولید می‌کند ، نیازی به بررسی شروط نیست.
گرامر LL(1) است

مفهوم اصلاح خطا یا Error Recovery :

برای اینکه کامپایلرها بتوانند حداکثر تعداد خطای برنامه را شناسایی کرده و گزارش دهند ، بایستی به نحوی از خطاهایی که شناسایی می‌کند ، چشم پوشی کند. برای این کار ، چند روش وجود دارد که بستگی به نوع تجزیه دارد.

روش‌های اصلاح خطا برای تجزیهٔ پیش‌نگر : ۱- روش اضطراری یا Panic Mode :

اکثر کامپایلرها از این روش استفاده می‌کنند و یا حداقل روش اصلی آنها برای اصلاح خطا ، همین روش است. با استفاده از این روش و در صورت بروز خطا ، خط تا یک نشانهٔ هماهنگی ، چشم پوشی می‌شود. این نشانه برای یک خط ، همان Semi Colon ؛ برای If ، بلوک If و برای حلقهٔ For ، کل حلقهٔ For است.

```
●●●;   // Error
…;
…;
For(…;…;…)
{
    …;
    …;
    …;
    if (●●●)
    {
        …;
        …;
        …;
    }
    …;
    …;
    …;
}
…;
…;
…;
```

⇒

```
●●●;   // ErrorRecovey
…;
…;
For(…;…;…)
{
    …;
    …;
    …;
    if (●●●)
    {
        …;
        …;
        …;
    }
    …;
    …;
    …;
}
…;
…;
…;
```

```
…;
…;
…;
For(…;●●●;…)
{
    …;
    …;
    …;
    if (…)
    {
        …;
        …;
        …;
    }
    …;
    …;
    …;
}
…;
…;
…;
```

⇒

```
…;
…;
…;
For(…;●●●;…)
{
    …;
    …;
    …;
    if (…)
    {
        …;
        …;
        …;
    }
    …;
    …;
    …;
}
…;
…;
…;
```

مجموعهٔ هماهنگ (S یا Synchronize) :

مجموعهٔ هماهنگ در جدول تجزیهٔ M برای هر متغیر ، عبارت است از مجموعهٔ $Fallow$ یا مجموعهٔ $First$ ، $Fallow$ آن متغیر. هر چه این مجموعه ، یک مجموعهٔ کاملتری باشد ، میزانی از برنامه که در صورت بروز خطا ، چشم پوشی خواهد شد ، کمتر است و خطای بیشتری در ابتدای کار مشخص خواهد شد.

مثال : مجموعهٔ هماهنگ در گرامر زیر که مربوط به مثال‌های قبلی هم هست را بدست آورید ؟

1. $E \to TE'$
2. $E' \to +TE'$
3. $E' \to \lambda$
4. $T \to FT'$
5. $T' \to *FT'$
6. $T' \to \lambda$
7. $F \to id$
8. $F \to (E)$

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

در خانه‌های بدست آمده در زیر ، S قرار می‌دهیم

$$Fallow(E) = \{\$,)\} \Rightarrow \begin{cases} M[E,)] \\ M[E,\$] \end{cases}$$

$$Fallow(T) = \{+,),\$\} \Rightarrow \begin{cases} M[T,+] \\ M[T,)] \\ M[T,\$] \end{cases}$$

$$Fallow(F) = \{+,*,),\$\} \Rightarrow \begin{cases} M[F,+] & M[F,)] \\ M[F,*] & M[F,\$] \end{cases}$$

چون در خانه‌های بدست آمده در زیر قانون است ، پس در آنها چیزی قرار نمی‌دهیم.

$Fallow(E') = \{\$,)\}$ ، $Fallow(T) = \{+,),\$\}$

الگوریتم اصلاح خطا در روش Panic Mode :

عملکرد این الگوریتم را به همراه مثال زیر مطرح می‌کنیم. فرض کنید می‌خواهیم در رشتهٔ $W =)id*+id\$$ اصلاح خطا انجام دهیم. خواهیم داشت :

Stack	Input	Action
$E	)id*+id$	Error , Delete ')'
$E	id*+id$	$E \to TE'$
$ET	id*+id$	$T \to FT'$
$ETF	id*+id$	$F \to id$
$ETid	id*+id$	Delete id
$ET'	*+id$	$T' \to *FT'$
$ETF*	*+id$	delete *
$ETF	+id$	Error , Delete 'F'
$ET'	+id$	$T' \to \lambda$
$E'	+id$	$E' \to +TE'$
$ET+	+id$	delete +
$ET	id$	$T \to FT'$
$ETF	id$	$F \to id$
$ETid	id$	Delete id
$ET'	$	$T' \to \lambda$
$E'	$	$E' \to \lambda$
$	$	Accept

مراحل الگوریتم به شرح زیر است :

۱. اگر $M[A,a] = S$ باشد ، متغیر بالای Stack را حذف می‌کنیم. البته به شرطی که تنها متغیر موجود در Stack نباشد. اگر تنها متغیر بالای Stack بود ، نماد ورودی را حذف می‌کنیم.

۲. اگر $M[A,a] = \varnothing$ باشد ، در اینصورت نماد ورودی ، یعنی a را حذف می‌کنیم.

۳. اگر نماد ورودی ≠ متغیر بالای Stack شد ، آنگاه متغیر بالای Stack را حذف می‌کنیم.

۲- روش Phraze Level :

این روش برعکس روش قبل ، بیشتر برنامه نویسی دارد. در این روش ، در هر خانهٔ خالی جدول ، اشاره‌گر به تابعی قرار دارد که اصلاح خطا در آن تابع انجام می‌گیرد. این توابع بررسی می‌کنند که به چه علت خطا رخ داده است و با تغییر ، حذف و یا درج یک نماد الفبا در ورودی یا بالای Stack ، باعث می‌شوند کامپایلر امکان ادامهٔ عمل کامپایل را برای شناسایی بقیهٔ خطاها داشته باشد.

تجزیه کنندگان پائین به بالا :

این تجزیه کنندگان ، عمومی‌تر از تجزیه کنندگان بالا به پائین هستند ؛ یعنی طیف بیشتری از گرامرها را تحت پوشش قرار می‌دهند که البته کمی پیچیده‌تر نیز هستند. توجه داشته باشید که تنها در صورتی به این نوع تجزیه کنندگان رجوع خواهیم کرد که گرامر LL(1) نباشد.

این نوع تجزیه کنندگان ، همانطور که در قبل نیز به آن اشاره شد ، شامل سه دسته هستند که عبارتند از :

۱. تجزیه کنندگان تقدم عملگر یا OP : که تنها بر روی گرامرهای عمل‌گرا قابل اعمال هستند. در این نوع تجزیه کنندگان ، جدول تجزیه‌ای وجود دارد که به آن ، جدول روابط تقدمی گفته می‌شود. این جدول با دو تابع FirstTerm() و LastTerm ساخته می‌شوند.

۲. تجزیه کنندگان تقدم ساده یا SP : که تنها بر روی گرامرهایی که فاقد قانون λ باشند ، قابل اعمال هستند. البته شرط دیگری نیز دارد که در جای خود توضیح داده خواهد شد. جدول مورد نیاز تجزیه در این روش نیز ، جدول روابط تقدکی نامیده می‌شود ولی با دو تابع Head() و Tail ساخته می‌شود.

۳. تجزیه کنندگان LR : که کامل‌ترین نوع تجزیه کننده‌ها بوده و قادر است هر نوع گرامر مستقل از متن را تجزیه کند. جدول تجزیه در این نوع تجزیه کننده با دو تابع First() و Fallow() به همراه یکسری توابع دیگر که در جای خود توضیح داده خواهند شد ، با سه روش قابل ساخت است :

a. Item LR(0) + SLR　　b. Item LR(1) + CLR　　c. LALR

نکته :

تفاوت جدول روابط تقدمی در روش OP و SP در این است که در روش تقدم عملگر ، در سطرها و ستون‌های این جدول ، نمادهای پایانی به همراه $ قرار می‌گیرند درحالیکه در روش تقدم ساده ، علاوه بر نمادهای پایانی و $ ، متغیرها نیز در سطر و ستون جدول ، قرار خواهد گرفت.

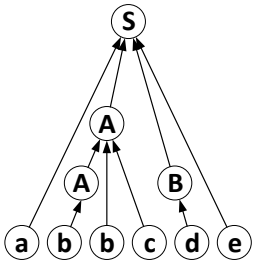
روش کلی تجزیه در این روش ، انتقال کاهشی یا Shift Reduce است که برعکس اشتقاق (بالا به پائین) که از ریشه به برگ می‌رسید ، از برگ به ریشه ختم می‌شود.

مثال : با توجه به گرامر زیر ، رشتهٔ مورد نظر را به روش پائین به بالا تجزیه نمائید :

$S \to aABe$
$A \to Abc \mid b$
$B \to b$

در واقع با این مثال می‌خواهیم روش Shift Reduce را به همراه عملکرد آن توضیح دهیم. از چپ به راست ، یکی یکی نمادهای ورودی را می‌خوانیم و در بین قانون‌ها ، به دنبال قانونی می‌گردیم که نماد خوانده شده در طرف سمت راست آن وجود داشته باشد.

- از ورودی a را می‌خوانیم. قانونی که سمت راست آن a باشد ، وجود ندارد.
- از ورودی b را می‌خوانیم. یک قانون وجود دارد که سمت راست آن b است. این قانون را به سمت چپ آن کاهش می‌دهیم. یعنی به جای سمت راست قانون ، سمت چپ آنرا قرار می‌دهیم. داریم : $W = aAbcde$
- مجدداً از اول رشته عملیات را تکرار می‌کنیم. $W = abbcde$
- از ورودی a را می‌خوانیم. قانونی که سمت راست آن a باشد ، وجود ندارد.
- از ورودی A را می‌خوانیم. قانونی که سمت راست آن aA باشد ، وجود ندارد.
  قانونی که سمت راست آن A باشد نیز ، وجود ندارد.
- از ورودی b را می‌خوانیم. یک قانون وجود دارد که سمت راست آن b است.
  داریم : $W = aAAcde$　　از ابتدا شروع می‌کنیم.
- ⋮
- از ورودی d را می‌خوانیم. یک قانون وجود دارد که سمت راست آن d است.
  داریم : $W = aAAcBe$　　از ابتدا شروع می‌کنیم.
- از ورودی a را می‌خوانیم. قانونی که سمت راست آن a باشد ، وجود ندارد.
- ⋮
- از ورودی e را می‌خوانیم. قانونی که سمت راست آن e باشد ، وجود ندارد. و ... پس یک مرحله را اشتباه رفتیم. به عقب بازگشته و آخرین کاهش را لغو کرده و سعی می‌کنیم قانون دیگری را پیدا کنیم.
- با یک مرحله بازگشت به جایی نمی‌رسیم. پس به دو مرحله قبل باز می‌گردیم که داشتیم : $W = aAbcde$. در این مرحله عمل Reduce یا کاهش درست نیست ؛ پس عمل انتقال یا Shift انجام می‌دهیم.
- از ورودی a را می‌خوانیم. قانونی نیست. از ورودی A را می‌خوانیم. قانونی نیست. از ورودی b را می‌خوانیم. قانونی نیست. از ورودی c را می‌خوانیم. یک قانون برای Abc وجود دارد. آنرا کاهش میدهیم. داریم : $W = aAde$　　از ابتدا شروع می‌کنیم.
- ⋮
- از ورودی d را می‌خوانیم. یک قانون وجود دارد که سمت راست آن d است. داریم : $W = aABe$　　از ابتدا شروع می‌کنیم.
- ⋮
- از ورودی e را می‌خوانیم. یک قانون وجود دارد که سمت راست آن $aABe$ است. پس از کاهش داریم : $W = S$



نکته :

همواره با خواندن هر نماد از ورودی ، سعی می‌کنیم بزرگ‌ترین رشته‌ای که از Stack می‌توان خواند و برای آن یک قانون یافت را انتخاب کنیم.

در تجزیه کنندگان پائین به بالا یا Shift Reduce ، دو عمل اصلی وجود دارد که عبارت است از :

۱. خواندن از ورودی و انتقال به Stack که به آن Shift گفته می‌شود.

۲. کاهش سمت راست یک قانون به سمت چپ آن که به آن Reduce گفته می‌شود.

برای تجزیه ، رشتهٔ ورودی از چپ به راست ، یکی یکی خوانده شده و درون Stack قرار می‌گیرد (Shift داخل Stack). تا زمانی که به یک دستگیره یا Handle برسیم. منظور از دستگیره ، شکل جمله‌ای سمت راست قوانین می‌باشد که این شکل جمله ، هر ترکیبی از متغیرها و پایانی‌ها می‌تواند باشد. با رسیدن به اولین دستگیره ، می‌توان عمل Reduce را انجام داد. در صورتی که عمل Reduce باعث شود که به جواب نرسیم ، بایستی مراحل را از نقطه‌ای که Reduce انجام شده ، مجدداً انجام دهیم ؛ ولی این مرتبه ، به جای عمل Reduce ، عمل Shift انجام می‌شود تا یک دستگیرهٔ دیگری برای عمل تجزیه پیدا شود. توجه داشته باشید که دستگیره ، طولانی‌ترین شکل جمله‌ای موجود در Stack خواهد بود.

پس ؛ مهمترین بخش در تجزیه‌های پائین به بالا ، انتخاب دستگیرهٔ درست می‌باشد. مشخص است که جداول در این روش ، بایستی عمل صحیح Shift یا Reduce را برای ما مشخص کنند.

نکته : در این نوع تجزیه کننده ، در ابتدا درون Stack ، $ قرار دارد و در انتها باید $ به همراه نماد شروع گرامر درون آن موجود باشد.

مثال : با توجه به گرامر داده شده ، رشتهٔ زیر را تجزیه نمائید :

$$W = id + id * id$$

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

Stack	Input	Action
$	$id + id * id \$$	Shift
$\$id$	$+id * id \$$	Reduce : 3
$\$E$	$+id * id \$$	Shift
$\$E +$	$id * id \$$	Shift
$\$E + id$	$*id \$$	Reduce : 3
$\$E + E$	$*id \$$	Shift (1)
$\$E + E *$	$id \$$	Shift
$\$E + E * id$	$\$$	Reduce : 3
$\$E + E * E$	$\$$	Reduce : 2
$\$E + E$	$\$$	Reduce : 1
$\$E$	$\$$	Accept

نکته (1) :

در این مرحله ، انتخاب صحیح توسط جدول تجزیه صورت می‌گیرد و لذا در اینجا بصورت ذهنی می‌توانیم بگوئیم که عمل Reduce درست نبوده و Shift درست است. اگر در اینجا عمل Reduce انجام می‌دادیم ، به جمع اولویتی بالاتر از ضرب داده بودیم که درست نیست.

نکته :

توجه داشته باشید که همواره دستگیره ، در بالای Stack قرار داشته و ظاهر خواهد شد که بایستی طولانی‌ترین رشته انتخاب شود.

تداخل‌ها در تجزیهٔ پائین به بالا :

وقتی در یک مرحله از تجزیه ، به جایی برسیم که ندانیم کدام عمل انتقال یا کاهش صحیح است ، تداخل رخ داده و انواع حالات آن عبارت است از :

۱. تداخل Shift-Reduce : مانند موردی که در مثال فوق رخ داد.

۲. تداخل Reduce-Reduce : زمانی رخ می‌دهد که دستگیرهٔ مورد نظر ، در بیش از یک قانون دیده شود. اگر در مثال فوق ، قانون $A \to id$ نیز وجود داشت ، این تداخل رخ میداد.

هر دو تداخل فوق ، با استفاده از جدول برطرف خواهند شد.

تجزیهٔ اولویتْ عملگر یا OP :

این تجزیه ، تنها روی گرامرهای عملگرا قابل اعمال است. این گرامرها دارای دو شرط به شرح زیر هستند :

۱. قوانین تولید λ ندارند.

۲. در سمت راست قوانین ، هیچگاه دو متغیر در کنار هم قرار نمی‌گیرند.

مانند گرامر در مثال فوق ($E \to E + E \mid E * E \mid id \mid (E)$)

جدول روابط تقدمی :

این جدول ، همانطور که از اسم آن مشخص است ، تقدم عملگرها که تنها در بین پایانی‌ها است را نشان می‌دهد.

در مثال زیر روابط تقدمی بین چند پایانی نمایش داده شده است.

- a : درون Stack و b : ورودی جاری است.
- $\boxed{\bullet>}$ نشاندهندهٔ این است که از سمت چپ ، عملگر +ی اول از عملگر +ی دوم اولویت بیشتری دارد.
- همواره اولویت id از تمامی عملگرها بیشتر است.
- همواره اولویت $\$$ از تمامی عملگرها کمتر است. یا به عبارتی تمامی پایانی‌ها اولویتی بیشتر از $\$$ دارند.
- عملگرهای ریاضی مطابق با اولویت خودشان هستند.
- خانه‌های خالی ، خطا هستند.

a \ b	id	$+$	$*$	$\$$
id	Error	$\bullet>$	$\bullet>$	$\bullet>$
$+$	$<\bullet$	$\boxed{\bullet>}$	$<\bullet$	$\bullet>$
$*$	$<\bullet$	$\bullet>$	$\bullet>$	$\bullet>$
$\$$	$<\bullet$	$<\bullet$	$<\bullet$	Accept

الگوریتم تجزیه به روش OP :

می‌خواهیم آنچه در مثال فوق اتفاق افتاد را بصورت الگوریتمی همراه با یک مثال بصورت مفصل توضیح دهیم.

مثال : رشتهٔ $id+id*id$ را به کمک جدول روابط تقدمی در مثال قبلی و گرامر داده شده ، به روش OP تجزیه نمائید.

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

$$W = id+id*id$$

Stack	Input	Action
$\$$	$id+id*id\$$	Shift
$\$id$	$+id*id\$$	Reduce : 3
$\$E$	$+id*id\$$	Shift
$\$E+$	$id*id\$$	Shift
$\$E+id$	$*id\$$	Reduce : 3
$\$E+E$	$*id\$$	Shift (1)
$\$E+E*$	$id\$$	Shift
$\$E+E*id$	$\$$	Reduce : 3
$\$E+E*E$	$\$$	Reduce : 2
$\$E+E$	$\$$	Reduce : 1
$\$E$	$\$$	Accept

با توجه به اینکه a درون Stack و b ورودی جاری است ، خواهیم داشت :

- اگر $a<\bullet b$ باشد ، در اینصورت ، عمل Shift یا انتقال انجام می‌شود.
- اگر $a\bullet>b$ باشد ، در اینصورت ، عمل Reduce یا کاهش انجام می‌شود. در واقع ، دستگیره در بالای Stack مشاهده خواهد شد.
- گر $a \doteq b$ باشد ؛ یعنی اولویت برابر باشد ؛ مانند پرانتز باز و بسته ؛ در اینصورت ، عمل Shift یا انتقال انجام می‌شود.

توجه داشته باشید که متغیرها در بالای Stack دیده نمی‌شوند ؛ برای مثال ، $\$E$ ، تنها $\$$ دیده می‌شود.

نکته (1) : در این حالت ، + و * دیده می‌شوند و در صورتی که جدول تجزیه وجود نداشت با مشکل مواجه می‌شدیم.

ایجاد جدول روابط تقدمی :

این جدول با استفاده از دو تابع $FirstTerm(A)$ و $LastTerm(A)$ که تنها برای متغیرها محاسبه می‌شوند ، ساخته خواهد شد.

تابع FirstTerm() :

خروجی این تابع برابر است با اولین ترمینالی که متغیر با آن شروع می‌شود. این متغیر ، همان متغیر ورودی تابع است.

$$FirstTerm(A)=\left\{a \mid A \overset{*}{\Rightarrow} a\alpha\right\}=\left\{a \mid A \overset{*}{\Rightarrow} Ba\alpha\right\}$$

تابع LastTerm() :

خروجی این تابع برابر است با آخرین ترمینالی که متغیر به آن ختم می‌شود. این متغیر ، همان متغیر ورودی تابع است.

$$LastTerm(A)=\left\{a \mid A \overset{*}{\Rightarrow} \alpha a\right\}=\left\{a \mid A \overset{*}{\Rightarrow} \alpha aB\right\}$$

نکته :

برای بدست آوردن جدول روابط تقدمی ، ابتدا باید یک گرامر غیر مبهم ایجاد کرد.

مثال : نتایج دو تابع فوق را برای کلیهٔ متغیرها در گرامر زیر بدست آورید :

$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to id \mid (E)$$

که رفع ابهام شدهٔ گرامر در مثال قبلی است

$$FirstTerm(E)=\{+,*,id,(\} \qquad LastTerm(E)=\{+,*,id,)\}$$
$$FirstTerm(T)=\{*,id,(\} \qquad LastTerm(T)=\{*,id,)\}$$
$$FirstTerm(F)=\{id,(\} \qquad LastTerm(F)=\{id,)\}$$

برای نمونه در تابع $FirstTerm(E)$ داریم :

$$E \Rightarrow E \underline{+} T \Rightarrow \cdots \quad , \quad E \Rightarrow T \Rightarrow T \underline{*} F \Rightarrow \cdots \quad , \quad E \Rightarrow T \Rightarrow F \Rightarrow \underline{id} \quad , \quad E \Rightarrow T \Rightarrow F \Rightarrow (\underline{E}) \Rightarrow \cdots$$

الگوریتم ایجاد جدول تقدمی :

که شامل چهار حالت به شرح زیر است : (a,b هر دو پایانی و W متغیر است)

۱. اگر قانونی بصورت $A \to \cdots aWb \cdots$ داشته باشیم ؛ مثل قانون $F \to (E)$ ، آنگاه در جدول داریم $a \doteq b$

۲. اگر قانونی بصورت $A \to \cdots aW \cdots$ داشته باشیم ؛ آنگاه در جدول داریم : a از تمامی $FirstTerm(W)$ اولویت کمتری دارد یعنی : $a \lessdot\bullet\ FirstTerm(W)$

۳. اگر قانونی بصورت $A \to \cdots Wb \cdots$ داشته باشیم ؛ آنگاه در جدول داریم : a از تمامی $LastTerm(W)$ اولویت بیشتری دارد یعنی : $LastTerm(W) \bullet\gtrdot b$

۴. اگر قانونی بصورت $A \to \cdots ab \cdots$ داشته باشیم ؛ بازهم اولویت آنها با هم برابر خواهد بود ، یعنی $a \doteq b$

مثال : ایجاد جدول روابط تقدمی برای گرامر زیر :

$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to id \mid (E)$$

➔

اولین گام در اینگونه مسائل
اضافه کردن یک قانون به ابتدای گرامر است
$$S \to \$E\$$$
به این گرامر جدید ، اصطلاحاً گرامر افزوده شده گفته می‌شود
این کار برای بدست آوردن روابط تقدمی مربوط به $ انجام می‌شود

➔

$$S \to \$E\$$$
$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to id \mid (E)$$

	id	$+$	$*$	$($	$)$	$\$$
id		$\bullet\gtrdot$	$\bullet\gtrdot$		$\bullet\gtrdot$	$\bullet\gtrdot$
$+$	$\lessdot\bullet$	$\bullet\gtrdot$	$\lessdot\bullet$	$\lessdot\bullet$	$\bullet\gtrdot$	$\bullet\gtrdot$
$*$	$\lessdot\bullet$	$\bullet\gtrdot$	$\bullet\gtrdot$	$\lessdot\bullet$	$\bullet\gtrdot$	$\bullet\gtrdot$
$($	$\lessdot\bullet$	$\lessdot\bullet$	$\lessdot\bullet$	$\lessdot\bullet$	$\doteq$	
$)$		$\bullet\gtrdot$	$\bullet\gtrdot$		$\bullet\gtrdot$	$\bullet\gtrdot$
$\$$	$\lessdot\bullet$	$\lessdot\bullet$	$\lessdot\bullet$	$\lessdot\bullet$		$\doteq Accept$

$$F \to (E) \Rightarrow \begin{cases} (\doteq) \\ (\lessdot\bullet FirstTerm(E) \\ LastTerm(E) \bullet\gtrdot) \end{cases} \qquad S \to \$E\$ \Rightarrow \begin{cases} \$ \doteq \$ \\ \$ \lessdot\bullet FirstTerm(E) \\ LastTerm(E) \bullet\gtrdot \$ \end{cases}$$

$$T \to T*F \Rightarrow \begin{cases} LastTerm(T) \bullet\gtrdot * \\ * \lessdot\bullet FirstTerm(F) \end{cases} \qquad E \to E+T \Rightarrow \begin{cases} LastTerm(E) \bullet\gtrdot + \\ + \lessdot\bullet FirstTerm(T) \end{cases}$$

هر خانه‌ای هم که خالی است ، خطا است.

روش‌های اصلاح خطا برای تجزیهٔ اولویتْ عملگر یا OP :

اصلاح خطا در این روش نیز ، همانند روش LL(1) ، به دو روش صورت می‌گیرد :

۱. Panic Mode یا اضطراری.
۲. Phraze Level

روش‌های اصلاح خطا برای تجزیهٔ اولویتْ عملگر یا OP : ۱- روش اضطراری یا Panic Mode :

در این نوع تجزیه هم ، یکسری چیزها حذف می‌شوند تا کامپایلر بتواند بقیهٔ خطاها را شناسایی کند.

مثال : می‌خواهیم خطاهای جدول تجزیهٔ داده شده در زیر را بدست آوریم :

	$+$	$($	$)$	$\$$	id
$\$$	$<\bullet$	$<\bullet$	$E1$	$\doteq$	$<\bullet$
$)$	$\bullet>$	$\bullet>$	$E2$	$E2$	$\bullet>$
id	$\bullet>$	$\bullet>$	$E2$	$E2$	$\bullet>$
$($	$<\bullet$	$<\bullet$	$\doteq$	$E3$	$<\bullet$
$+$	$\bullet>$	$\bullet>$	$<\bullet$	$<\bullet$	$<\bullet$

ابتدا بررسی می‌کنیم که به چه علت در خانه‌های خالی خطا رخ داده است. خواهیم داشت :

- $E1$: بالای Stack خالی شده ولی در ورودی ، پرانتز بسته وجود دارد. پس یک پرانتز بسته در ورودی اضافه است که کامپایلر برای ادامه دادن ، آنرا حذف می‌کند.
- $E2$: همواره بایستی در بین دو عملوند ، از یک عملگر استفاده شود ؛ ولی در این حالت ، عملگری وجود ندارد. در این حالت پیغام خطای *Missing Operator* داده خواهد شد. از طرفی برای چشم پوشی این خطا ، اینبار کامپایلر به جای حذف یکسری اطلاعات ، یک عملگر در جای تشخیص داده شده اضافه شده و در نتیجه به ادامهٔ خطایابی می‌پردازد.
- $E3$: در این حالت ، ورودی به پایان رسیده ولی هنوز در Stack ، پرانتز باز وجود دارد. یعنی یک پرانتز بسته کم نوشته شده است. در این حالت ، پرانتز باز از بالای Stack حذف شده و بدینوسیله خطا پوشش داده خواهد شد

در این روش ، تنها یکسری خطاها و آنهم خطاهای نوع اول که مربوط به خانه‌های خالی در جدول است ، تشخیص داده می‌شوند.

۲- روش Phraze Level :

در این روش ، خطاهای نوع دوم بررسی و پوشش داده خواهند شد. این نوع خطاها زمانی رخ می‌دهند که دستگیرهٔ مشاهده شده در بالای Stack ، در سمت راست هیچ قانونی وجود نداشته باشد. بسته به این نوع خطا ، سه حالت بوجود خواهد آمد که عبارتند از :

توضیح	پیغام خطا	شبیه‌ترین سمت راست قانون	دستگیرهٔ بالای Stack
نام متغیرها در این مرحله مهم نیست پس در اینجا یک b اضافه نوشته شده است	*illegal 'b' in Line ...*	aDc	$aNbc$
یک c کم نوشته شده است	*missing 'c' in Line ...*	$abNcd$	$abAd$
معمولاً فرض می‌کند N در نهایت با یک عملگر تعویض خواهد شد	*missing operator in Line ...*	$aNbc$	abc

همانطور که در بالا هم مشاهده می‌شود ، ممکن است پیغام‌های خطا اشتباه داده شوند ؛ یعنی در واقع این پیغام‌ها ، دقیق نیستند. پس روش دقیقی برای اعلام خطا نیست. البته تجزیهٔ اولویت عملگر ، سرعت تجزیهٔ بالایی نسبت به دیگر تجزیه کنندگان پائین به بالا دارد.

تجزیه به روش تقدم ساده یا SP :

در واقع این روش ، تعمیم یافتهٔ روش قبلی است ؛ یعنی گرامرهای بیشتری را تحت پوشش قرار می‌دهد. اگر گرامر تنها قانون λ نداشته باشد ، با این روش قابل تجزیه خواهد بود. جدول تقدمی در این نوع تجزیه ، همانطور که قبلاً هم به آن اشاره شد ، هم بین متغیرها و هم بین پایانی‌ها در نظر گرفته خواهد شد.

ایجاد جدول روابط تقدمی در تجزیهٔ تقدم ساده :

جدول تجزیه در این روش ، با دو تابع بدست خواهد آمد که هر دوی آنها بر روی متغیرها تعریف می‌شوند و عبارتند از :

۱. $Head(A)$: شامل تمامی نمادهایی (چه پایانی و چه متغیر) که A با آن شروع می‌شود.

$$Head(A)=\left\{X \quad | \quad A \overset{*}{\Rightarrow} X\,\alpha\right\}$$

که در آن X ، یک پایانی یا یک متغیر است

۲. $Tail(A)$: شامل تمامی نمادهایی (چه پایانی و چه متغیر) که A با آن ختم می‌شود.

$$Head(A)=\left\{X \quad | \quad A \overset{*}{\Rightarrow} \alpha X\right\}$$

که در آن X ، یک پایانی یا یک متغیر است

ولی چطور این جدول ایجاد می‌شود ؟

الگوریتم ایجاد جدول روابط تقدمی در تجزیهٔ تقدم ساده :

که شامل چهار حالت به شرح زیر است : (a,b هر دو پایانی و A,B هر دو متغیر هستند)

۱. اگر قانونی بصورت $U \rightarrow \cdots ab \cdots$ داشته باشیم ؛ آنگاه در جدول داریم $a \doteq b$

۲. اگر قانونی بصورت $U \rightarrow \cdots aB \cdots$ داشته باشیم ؛ آنگاه به دو نتیجه در جدول می‌رسیم :

a. $a \doteq B$

b. $a \lessdot \bullet Head(B)$

۳. اگر قانونی بصورت $U \rightarrow \cdots Ab \cdots$ داشته باشیم ؛ آنگاه به دو نتیجه در جدول می‌رسیم :

a. $A \doteq b$

b. $Tail(A) \bullet\!> b$

۴. اگر قانونی بصورت $U \rightarrow \cdots AB \cdots$ داشته باشیم ؛ آنگاه به چهار نتیجه در جدول می‌رسیم :

a. $A \doteq B$

b. $A \lessdot \bullet Head(B)$

c. $Tail(A) \bullet\!> B$

d. $Tail(A) \bullet\!> Head(B)$

مثال : ایجاد جدول روابط تقدمی برای گرامر زیر :

$$\begin{matrix} S \rightarrow (SS) \\ S \rightarrow c \end{matrix} \quad \rightarrow$$

اولین گام در اینگونه مسائل
اضافه کردن یک قانون به ابتدای گرامر است
$N \rightarrow \$S\,\$$

$$\rightarrow \quad \begin{matrix} N \rightarrow \$S\,\$ \\ S \rightarrow (SS) \\ S \rightarrow c \end{matrix}$$

	S	(	)	c	$\$$
S	$\doteq$	$\lessdot\bullet$	$\doteq$	$\lessdot\bullet$	$\doteq$
(	$\doteq$	$\lessdot\bullet$	E	$\lessdot\bullet$	E
)	$\bullet\!>$	$\bullet\!>$	$\bullet\!>$	$\bullet\!>$	$\bullet\!>$
c	$\bullet\!>$	$\bullet\!>$	$\bullet\!>$	$\bullet\!>$	$\bullet\!>$
$\$$	$\doteq$	$\lessdot\bullet$	E	$\lessdot\bullet$	E

$$N \rightarrow \$S\,\$ \Rightarrow \begin{cases} \$S \Rightarrow \begin{cases} \$ \doteq S \\ \$ \lessdot \bullet Head(S) \quad \Rightarrow \quad \$ \lessdot \bullet \{(,c\} \end{cases} \\ S\,\$ \Rightarrow \begin{cases} S \doteq \$ \\ Tail(S) \bullet\!> \$ \quad \Rightarrow \quad \{(,c\} \bullet\!> \$ \end{cases} \end{cases}$$

$$S \rightarrow (SS) \Rightarrow \begin{cases} (S \Rightarrow \begin{cases} (\doteq S \\ (\lessdot \bullet Head(S) \quad \Rightarrow \quad (\lessdot \bullet \{(,c\} \end{cases} \\ SS \Rightarrow \begin{cases} S \doteq S \\ S \lessdot \bullet Head(S) \quad \Rightarrow \quad S \lessdot \bullet \{(,c\} \\ Tail(S) \bullet\!> S \quad \Rightarrow \quad \},c\} \bullet\!> S \\ Tail(S) \bullet\!> Head(S) \quad \Rightarrow \quad \},c\} \bullet\!> \{(,c\} \end{cases} \\ S) \Rightarrow \begin{cases} S \doteq) \\ Tail(S) \bullet\!>) \quad \Rightarrow \quad \},c\} \bullet\!>) \end{cases} \end{cases}$$

الگوریتم تجزیه به روش تقدمی ساده :

الگوریتم تجزیه را همراه با یک مثال توضیح خواهیم داد :

مثال : رشتهٔ $(c(cc))$ را به کمک جدول روابط تقدمی در مثال قبلی و گرامر داده شده ، به روش SP تجزیه نمائید.

$$\left.\begin{array}{ll} S \to (SS) & 1 \\ S \to c & 2 \end{array}\right\} \quad W = (c(cc))$$

Stack	Input	Action
$	$(c(cc))$	Shift
$ <•(	$c(cc))$	Shift
$ <•(<•c	$(cc))$	Reduce : 2
$ <•(S	$(cc))$	Shift
$ <•(S <•(	$cc))$	Shift
$ <•(S <•(<•c	$c))$	Reduce : 2
$ <•(S <•(S	$c))$	Shift
$ <•(S <•(S <•c	$))$	Reduce : 2
$ <•(S <•(SS	$))$	Shift
$ <•(S <•(SS)	$)$	Reduce : 1
$ <•(SS	$)$	Shift
$ <•(SS)	$	Reduce : 1
$S	$	Accept

با توجه به اینکه a درون Stack و b ورودی جاری است ، خواهیم داشت :

- اگر $a <• b$ باشد ، در اینصورت ، عمل Shift یا انتقال انجام شده و هم $<•$ و هم b داخل Stack قرار خواهد گرفت.
- اگر $a •> b$ باشد ، در اینصورت ، عمل Reduce یا کاهش انجام می‌شود. توجه داشته باشید که تا علامت $<•$ داخل Stack ، دستگیره خواهد بود. پس از عمل Reduce ، آخرین نماد بالای Stack و متغیر وارد شده به آن ، در جدول تجزیه بررسی می‌شوند. اگر اولویت نماد بالای Stack از متغیر وارد شده به آن کمتر باشد ، بین این دو ، علامت $<•$ قرار خواهد گرفت. در غیر اینصورت ، چیزی درج نخواهد شد.
- اگر $a \doteq b$ باشد ؛ عمل Shift انجام و تنها b داخل Stack قرار خواهد گرفت.

روش‌های اصلاح خطا برای تجزیهٔ تقدمی ساده :

تصحیح خطا در این نوع تجزیه ، همانند روش قبلی است. (به روش قبلی مراجعه نمائید)

تجزیه به روش LR :

حرف L در این روش ، نشانگر خواندن نمادهای ورودی از چپ به راست و حرف R ، به علت Right Most بودن است.

تجزیه کننده‌های LR ، یک روش کلی برای تجزیهٔ تمامی گرامرهای مستقل از متن می‌باشند. این تجزیه کننده‌ها ، روال پیچیده‌تری برای ایجاد جداول تجزیه دارند.

نکته :

تجزیه‌های پائین به بالا ، برعکس تجزیهٔ بالا به پائینی هستند که به روش Right Most انجام شده است. برای نشان دادن این قضیه ، به مثال پائین توجه نمائید :

مثال : بدست آوردن تجزیهٔ بالا به پائین و اشتقاق Right Most در رشتهٔ $W = aababbb$ و با توجه به گرامر زیر :

$$S \to aSAb \mid B \mid aSb$$
$$A \to aA \mid b$$
$$B \to b$$

تجزیهٔ پائین به بالا : $aa\underline{b}abbb \Rightarrow aa\underline{B}abbb \Rightarrow aaSa\underline{b}bb \Rightarrow aaS\,\underline{aAb}b \Rightarrow a\underline{aSAb}b \Rightarrow \underline{aSb} \Rightarrow S$

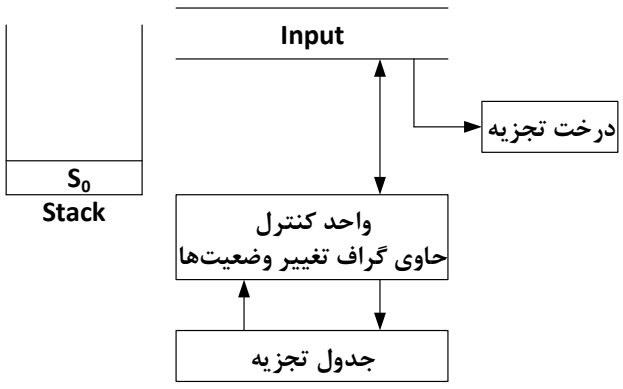
اشتقاق Right Most : $S \Rightarrow aSb \Rightarrow aaSAbb \Rightarrow aaSaAbb \Rightarrow aaSabbb \Rightarrow aaBabbb \Rightarrow aababbb$

در صورتیکه اشتقاق Right Most را برعکس کنیم ، تجزیهٔ پائین به بالا خواهیم داشت.

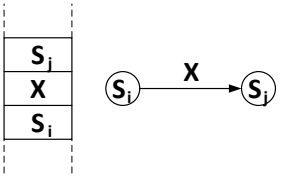
نکتهٔ فوق ، نشان می‌دهد که این تجزیه کننده‌ها هیچ محدودیتی ندارند و تمامی گرامرها را تحت پوشش قرار می‌دهند.

عملکرد تجزیه کننده‌های LR :

عملکرد این تجزیه کننده نیز ، مانند سایر PDA ها است با چند تفاوت جزئی :



Stack حاوی شمارهٔ وضعیت‌ها و نمادهای گرامر است که البته در ابتدا ، تنها وضعیت اول ، یعنی S_0 داخل آن قرار دارد. همواره محتوای Stack بصورت زیر است : (X هم می‌تواند پایانی و هم می‌تواند متغیر باشد)



و این نشان می‌دهد که از وضعیت S_i با نماد X به وضعیت S_j خواهیم رفت

جدول تجزیه هم کمی متفاوت است :

در سطرهای جدول ، وضعیت‌های واحد کنترل یا همان وضعیت‌های PDA قرار می‌گیرند.

ستون‌ها نیز به دو بخش تقسیم می‌شوند :

- بخش Action : که در آن پایانی‌ها و $ قرار می‌گیرند
- بخش GoTo : که در آن متغیرها قرار می‌گیرند

	Action	GoTo
S_0		
S_1		
⋮		

همانطور که قبلاً هم اشاره شد ، جدول تجزیه به سه روش قابل ساخت است که البته عملکرد تجزیه با آنها تفاوتی نخواهد کرد. این سه روش عبارتند از :

۱. SLR : در این روش ، جدول تجزیه سریع ساخته خواهد شد ؛ ولی سرعت تجزیه توسط آن کم است. این روش با استفاده از مفهوم جدیدی به نام Item LR(0) انجام می‌شود.

۲. CLR : جدول ساخته شده در این روش ، بزرگتر از جدول ساخته شده در مدل قبلی است ولی سرعت تجزیه توسط آن بیشتر از سرعت در مدل قبلی است. این روش نیز با استفاده از مفهوم جدیدی به نام Item LR(1) انجام می‌شود.

۳. LALR : در این روش ، جدول بدست آمده از مدل CLR خلاصه خواهد شد. پس جدول خلاصه‌تر و کوچکتری داریم ولی سرعت آن حفظ می‌شود. این خلاصه سازی نیز از ادغام سطرها بوجود می‌آید.

الگوریتم تجزیه به روش LR :

فعلاً فرض می‌کنیم جدول تجزیه را در اختیار داریم و تنها می‌خواهیم الگوریتم تجزیه را شرح دهیم. این الگوریتم را به همراه یک مثال مطرح می‌کنیم :

مثال : با توجه به جدول تجزیه و گرامر داده شده در زیر ، رشتهٔ $W = id * id + id$ را تجزیه نمائید.

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

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id \mid (E)$

برای درک بهتر ، گرامر را بصورت زیر بازنویسی کرده و به هر قانون شماره‌ای را اختصاص می‌دهیم. خواهیم داشت :

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

نکات جدول تجزیه :

- اعدادی که در زیر ستون‌های مربوط به GoTo نوشته می‌شود ، شمارهٔ وضعیت‌ها هستند.
- منظور از $\boxed{S_5}$ وضعیت پنجم نبوده و منظور $Shift$ پنجم است.
- منظور از $\boxed{R_2}$ ، $Reduce$ با استفاده از قانون پنجم است.

Stack	Input	Action	
0	$id*id+id\$$	S_5	
$0id\,5$	$*id+id\$$	R_6	$F \to id$
$0F\,3$	$*id+id\$$	R_4	$T \to F$
$0T\,2$	$*id+id\$$	S_7	
$0T\,2*7$	$id+id\$$	S_6	
$0T\,2*7id\,5$	$+id\$$	R_6	$F \to id$
$0T\,2*7F\,10$	$+id\$$	R_3	$T \to T*F$
$0T\,2$	$+id\$$	R_2	$E \to T$
$0E\,1$	$+id\$$	S_6	
$0E\,1+6$	$id\$$	S_7	
$0E\,1+6id\,5$	$\$$	R_6	$F \to id$
$0E\,1+6F\,3$	$\$$	R_4	$T \to F$
$0E\,1+6T\,9$	$\$$	R_1	$E \to E+T$
$0E\,1$	$\$$	$\Rightarrow$	$Accept$

نکات تجزیه :

- اعداد در Stack نشانگر وضعیت هستند.
- در هر بار $Reduce$ ، به اندازهٔ دو برابر سمت راست قانون را از Stack برداشته و به جای آن ، سمت راست قانون را قرار می‌دهیم. همچنین پس از قرار دادن سمت چپ ، عددی که در سطر وضعیت و متغیر در جدول تجزیه است را در بالای Stack قرار می‌دهیم. برای مثال در سطر دوم ، id و 5 برداشته شده و به جای آن F قرار داده می‌شود. همچنین $Goto(0,F)$ نیز در بالای Stack قرار می‌گیرد.
- در هر بار $Shift$ نیز ، علاوه بر درج متغیر یا پایانی در Stack ، عدد مربوط به Shift در جدول تجزیه نیز درج می‌شود

ایجاد جدول تجزیهٔ LR به روش SLR :

قبل از اینکه روش ایجاد این جدول توضیح داده شود ، نیاز به تعریف چند مفهوم داریم.

تعریف Item LR(0) : اگر قانونی به شکل $A \to XYZ$ داشته باشیم ، Item LR(0)های این قانون ، قانون‌های زیر خواهد بود :

$A \to \bullet XYZ \quad , \quad A \to X \bullet YZ \quad , \quad A \to XY \bullet Z \quad , \quad A \to XYZ \bullet$

پس این تعریف ، یک نقطه‌ای است که در هر جای قانون می‌تواند باشد. ولی چه چیزی را مشخص می‌کند ؟ (در آینده خواهیم دید)

تعریف $Closure$ یا بستار Item LR(0) : اگر قوانینی به شکل $B \to \gamma$, $A \to \alpha \bullet B\beta$ داشته باشیم ؛ یعنی نقطه در قبل از یک متغیر واقع شده باشد ، بستار Item LR(0) قانون A ، قانون‌های $B \to \bullet\gamma$, $A \to \alpha \bullet B\beta$ خواهد بود ؛ یعنی خودش به همراه تمامی قوانینی که نقطه قبل از قانون B آمده است. توجه داشته باشید که اگر با نوشتن B ، باز هم نقطه قبل از متغیر دیگری آمد ، برای آن متغیر هم همین عمل را تکرار می‌کنیم (یعنی بصورت بازگشتی).

مثال : بستار Item LR(0) در قانون $E \to \bullet E + T$ در گرامر زیر را بدست آورید :

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

$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet T * F$
$T \to \bullet F$
$F \to \bullet (E)$
$F \to \bullet id$

پس در مجموع خواهیم داشت

➔

$F \to \bullet (E)$
$F \to \bullet id$

تقطه پشت متغیر است F هم پدیدار شد

➔

$T \to \bullet T * F$
$T \to \bullet F$

تقطه پشت متغیر است T هم پدیدار شد

➔

$E \to \bullet E + T$
$E \to \bullet T$

گرامر SLR(1) :

گرامری SLR(1) است که در تجزیهٔ جدول SLR آن ، خانه‌ای با دو مقدار نداشته باشیم. (راه دیگری هم برای یافتن آن وجود ندارد.)

رسم دیاگرام SLR :

برای ایجاد جدول تجزیه ، ابتدا بایستی یک دیاگرام SLR تشکیل داد. این دیاگرام ، یک نمودار تبدیل وضعیت یا Transition Diagram است که توسط آن ، جدول تجزیه ایجاد خواهد شد. روش ایجاد این دیاگرام را همراه با یک مثال توضیح خواهیم داد.

مثال : دیاگرام SLR ، مربوط به گرامر زیر را رسم کنید.

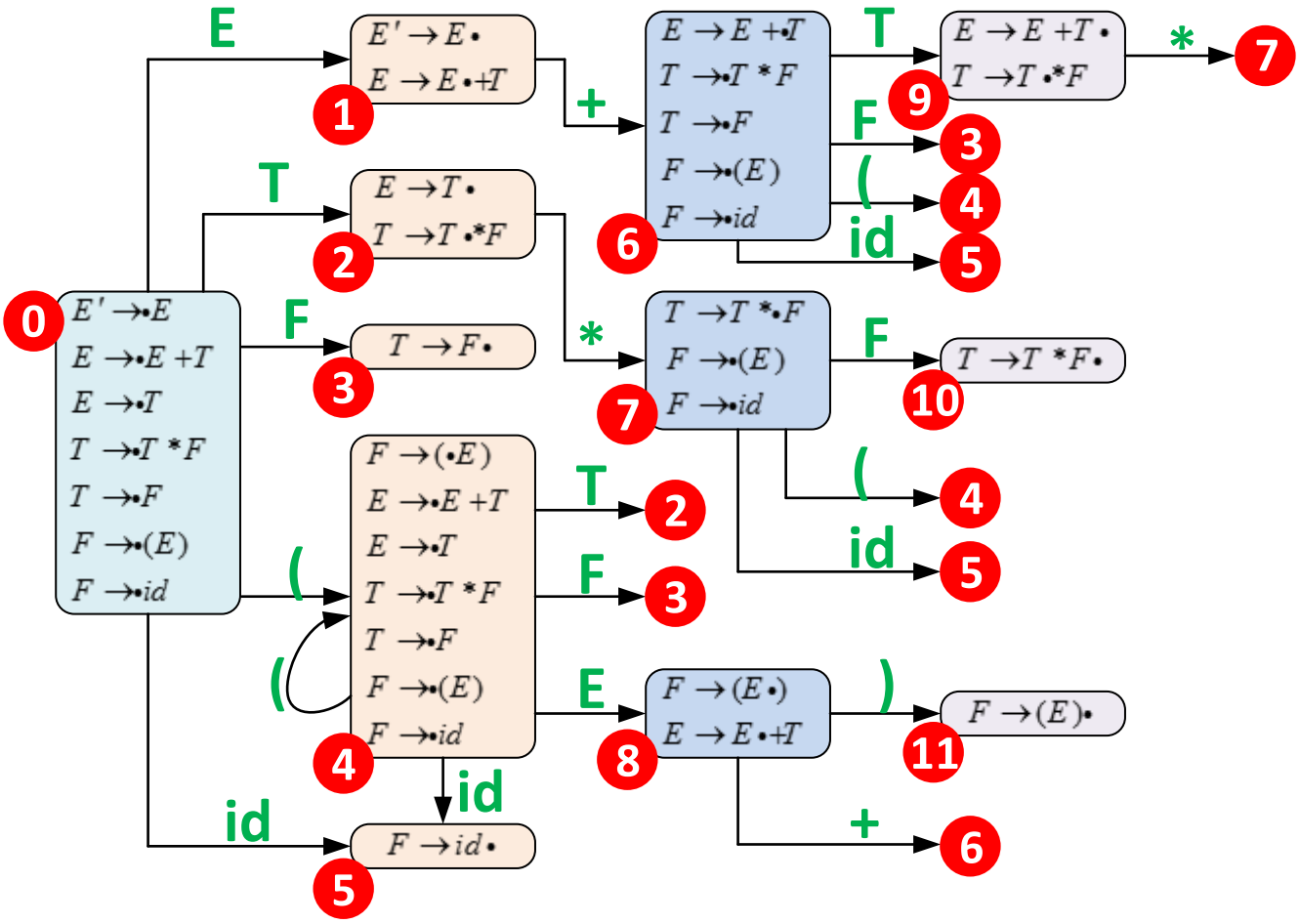
$$
\begin{array}{ll}
1 & E \to E+T \\
2 & E \to T \\
3 & T \to T*F \\
4 & T \to F \\
5 & F \to (E) \\
6 & F \to id
\end{array}
$$

در نهایت با تفکیک هر قانون و شماره گذاری آنها خواهیم داشت

$$
\begin{array}{l}
E' \to E \\
E \to E+T \mid T \\
T \to T*F \mid F \\
F \to (E) \mid id
\end{array}
$$

در اولین گام ، یک قانون جدید به آن اضافه می‌کنیم

$$
\begin{array}{l}
E \to E+T \mid T \\
T \to T*F \mid F \\
F \to (E) \mid id
\end{array}
$$



مراحل کار بصورت زیر است :

۱. در ابتدای کار ، همانطور که قبلاً هم به آن اشاره شد ، قانون جدیدی به گرامر اضافه می‌کنیم.
۲. از قانون اول شروع کرده ، قانون را بازنویسی می‌کنیم و قبل از اولین متغیر یا پایانی در سمت راست قانون ، یک نقطه قرار می‌دهیم.
۳. با نوشتن هر قانون ، اگر نقطه قبل از متغیری قرار گرفت ، بستار آن متغیر را نیز در بازنویسی می‌کنیم.
۴. کل مجموعهٔ بدست آمده را یک وضعیت در نظر گرفته و به آن یک شماره اختصاص می‌دهیم.
۵. سپس به وضعیت نگاه کرده و متغیر یا پایانی‌هایی که قبل از نقطه قرار گرفته‌اند را مشخص کرده و به ازای هر کدام یک خروجی در نظر می‌گیریم.
۶. به ازای تمامی قوانینی که این متغیر جزء آنهاست ، قوانین را نوشته و نقطه را یکی به جلو می‌بریم.
۷. اگر نقطه قبل از متغیری قرار گرفت ، بستار آن متغیر را نیز در بازنویسی می‌کنیم.
۸. به مرحلهٔ چهار رفته و الگوریتم را تکرار می‌کنیم تا زمانیکه وضعیت جدیدی بدست نیاید.

ایجاد جدول تجزیه از روی دیاگرام SLR :

مراحل بصورت زیر است :

۱. در هر وضعیت ، به ازای خروجی متغیرها ، تنها وضعیت جدید را در جدول می‌نویسیم. همچنین به ازای خروجی پایانی‌ها ، S و شمارهٔ وضعیت مقصد ، نوشته خواهد شد. برای نمونه در مثال صفحهٔ قبل ، دیدیم که از وضعیت صفر با متغیر E به وضعیت اول رفتیم. پس در جدول ، در سطر مربوط به وضعیت صفر و در ستون E ، شمارهٔ وضعیت جدید ، یعنی 1 می‌نویسیم همچنین از وضعیت صفر با پایانی id ، به وضعیت پنجم رفتیم. پس در جدول ، در سطر مربوط به وضعیت صفر و در ستون id ، S_5 می‌نویسیم که به معنی $Shift$ و تغییر وضعیت به وضعیت پنجم می‌باشد.

۲. ولی $Reduce$ چی ؟ در وضعیت‌های بدست آمده ، کلیهٔ وضعیت‌هایی که با نقطه ختم شده‌اند را می‌یابیم. به ازای هر قانونی بصورت $A \to \alpha\bullet$ به شکل زیر عمل می‌کنیم : در جدول $M[\ , b] = R$ شمارهٔ وضعیت قرار می‌دهیم که $b \in Fallow(A)$. اندیس R را نیز شمارهٔ قانون موجود در گرامر می‌گذاریم.

a. اگر اندیس R ، صفر بدست آمد ، در جدول $M[\ , b] = Acc$ شمارهٔ وضعیت قرار می‌دهیم.

برای نمونه ، در مثال صفحهٔ قبل خواهیم داشت :

$$State : 2 \Rightarrow E \to T\bullet \Rightarrow Fallow(E) = \{\$,\ +,\ )\} \Rightarrow M[2,\$] = M[2,+] = M[2,)] = R_2$$

$$State : 3 \Rightarrow T \to F\bullet \Rightarrow Fallow(T) = \{\$,\ +,\ ),\ *\} \Rightarrow M[3,\$] = M[3,+] = M[3,)] = M[3,*] = R_4$$

$$State : 5 \Rightarrow F \to id\bullet \Rightarrow Fallow(F) = \{\$,\ +,\ ),\ *\} \Rightarrow M[5,\$] = M[5,+] = M[5,)] = M[5,*] = R_6$$

$$State : 9 \Rightarrow E \to E+T\bullet \Rightarrow Fallow(E) = \{\$,\ +,\ )\} \Rightarrow M[9,\$] = M[9,+] = M[9,)] = R_1$$

$$State : 10 \Rightarrow T \to T*F\bullet \Rightarrow Fallow(T) = \{\$,+,),*\} \Rightarrow M[10,\$] = M[10,+] = M[10,)] = M[10,*] = R_3$$

$$State : 11 \Rightarrow T \to (E)\bullet \Rightarrow Fallow(F) = \{\$,+,),*\} \Rightarrow M[11,\$] = M[11,+] = M[11,)] = M[11,*] = R_5$$

مثال : نشان دهید گرامر زیر ، یک گرامر SLR(1) نیست :

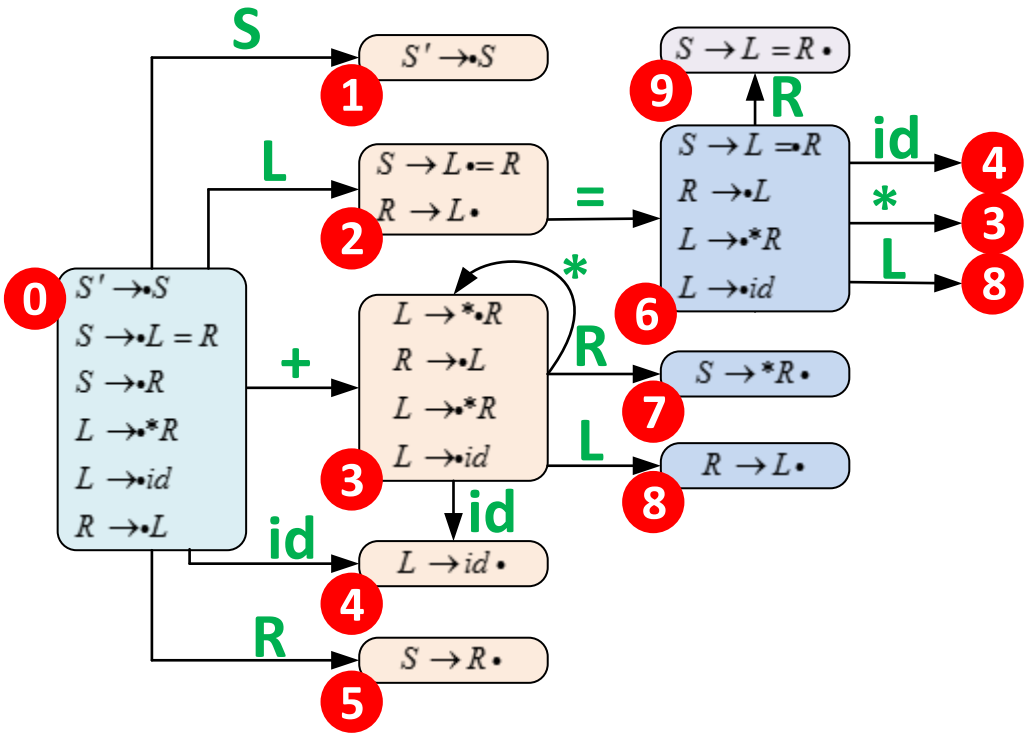
$$S \to L = R \mid R$$
$$L \to {*}R \mid id$$
$$R \to L$$

➔ در اولین گام ، یک قانون جدید به آن اضافه می‌کنیم ➔

$$S' \to S$$
$$S \to L = R \mid R$$
$$L \to {*}R \mid id$$
$$R \to L$$

➔ در نهایت با تفکیک هر قانون و شماره گذاری آنها خواهیم داشت ➔

1. $S \to L = R$
2. $S \to R$
3. $L \to {*}R$
4. $L \to id$
5. $R \to L$



	+	id	=	$	S	L	R
S_0	S_3	S_4			1	2	3
S_1				Acc			
S_2			~~S_6, R_5~~	~~R_2, R_5~~			
S_3	S_3	S_4				8	7
S_4							
S_5				R_2			
S_6	S_3	S_4				8	9
S_7							
S_8							
S_9							

$$State : 2 \Rightarrow S \to R\bullet \Rightarrow Fallow(S) = \{\$\} \Rightarrow M[2,\$] = R_2$$
$$State : 2 \Rightarrow R \to L\bullet \Rightarrow Fallow(R) = \{\$, =\} \Rightarrow M[2,\$] = M[2,=] = R_5$$

رسم نمودار تبدیل وضعیت CLR :

قبل از اینکه روش ایجاد این نمودار توضیح داده شود ، نیاز به تعریف چند مفهوم داریم.

تعریف Item LR(1) : در اینجا ، یک چیزی اضافه‌تر از Item LR(0) داریم. خواهیم داشت :

$$Item\ \ LR(1) = \{A \to \alpha \cdot B\,\beta \quad , \quad \{a\}\}$$

که در آن $A \to \alpha \cdot B\,\beta$ همان Item LR(0) و به $\{a\}$ ، مجموعهٔ پیش‌نگر یا Look Ahead (LA) گفته می‌شود که داریم :

$$LA = Fallow\,(B)$$

و از قبل هم می‌دانیم که :

$$If \quad \beta = \lambda \quad \Rightarrow \quad LA = Fallow\,(A) \quad or \quad \{A \to \alpha \cdot B \quad , \quad \{a\}\}$$

تعریف $Closure$ یا بستار Item LR(1) : این بستار در حالت کلی شامل تمامی قوانینی است که با B شروع می‌شوند و سمت راست آنها در ابتدای کار نقطه قرار می‌گیرد ، به همراه مجموعهٔ LAی که از $First\,(\beta a)$ بدست می‌آید. یعنی : $\{B \to \cdot\gamma \quad , \quad \{First\,(\beta a)\}\}$

روش ایجاد نمودار CLR را به همراه با یک مثال توضیح خواهیم داد.

مثال : دیاگرام CLR(1) ، مربوط به گرامر زیر را رسم کنید.

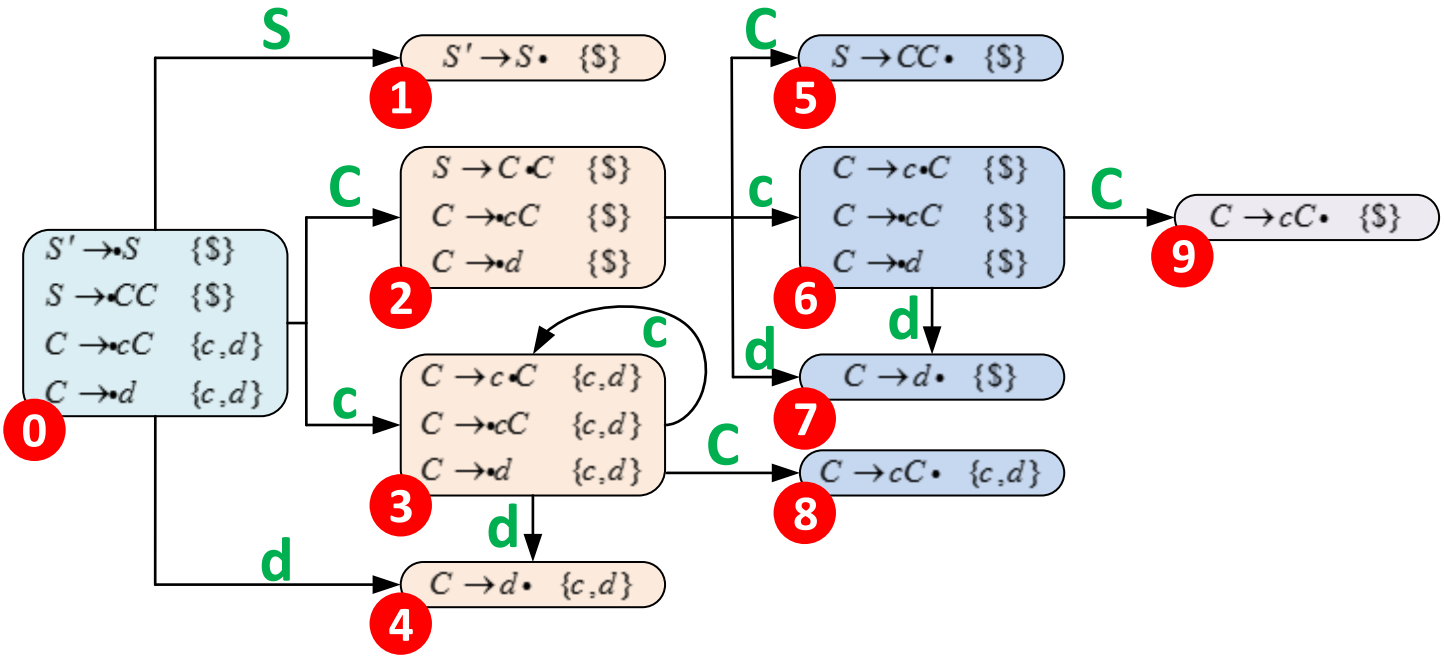
$$S \to CC$$
$$S \to cC \mid d$$

→ در اولین گام ، یک قانون جدید به آن اضافه می‌کنیم →

$$S' \to S$$
$$S \to CC$$
$$S \to cC \mid d$$

→ در نهایت با تفکیک هر قانون و شماره گذاری آنها خواهیم داشت →

1. $S \to CC$
2. $S \to cC$
3. $S \to d$



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

تنها تفاوت با SLR در این است که برای Reduceها ، به ازای قوانینی که در انتهای آن ، نقطه قرار دارد ، به ازای شمارهٔ وضعیت و تمامی عناصر موجود در مجموعهٔ پیش‌نگر ، Reduce با شمارهٔ قانون را قرار می‌دهیم.

گرامر CLR(1) :

گرامری CLR(1) است که در تجزیهٔ جدول CLR آن ، خانه‌ای با دو مقدار نداشته باشیم ؛ یعنی در هر خانه ، تنها یک قانون وجود داشته باشد.

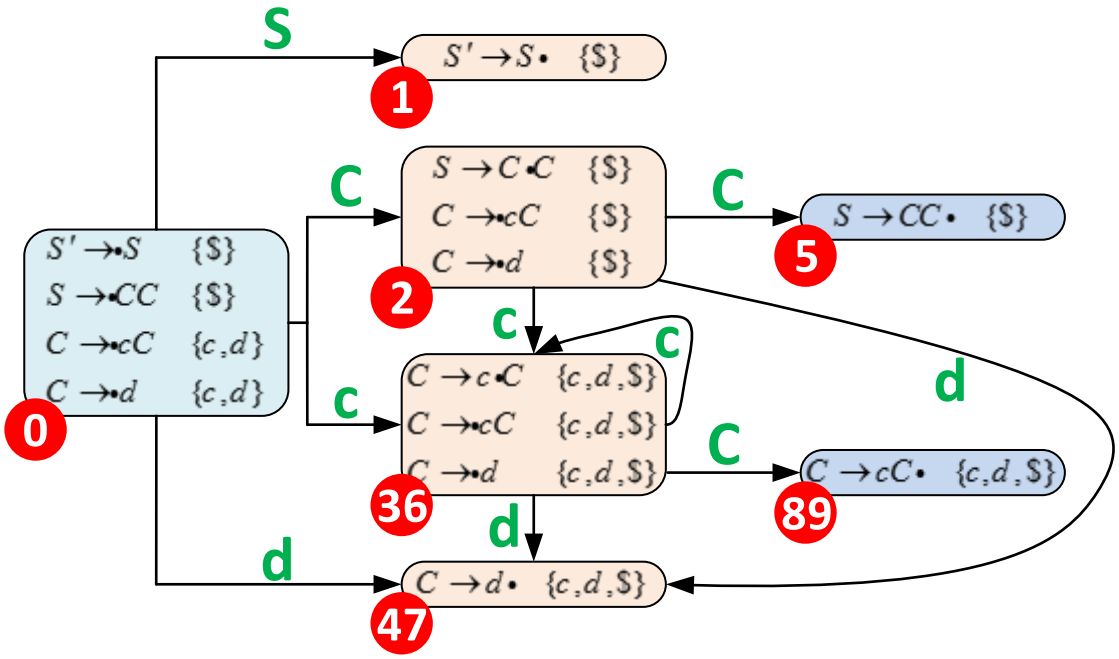
نحوهٔ ایجاد دیاگرام LALR :

برای رسم این دیاگرام ، ابتدا دیاگرام CLR را رسم کرده ، سپس تمامی وضعیت‌هایی که دارای بخش Item LR(0) یکسان هستند را در هم ادغام کرده ، مجموعهٔ پیش‌نگر آنها را نیز بصورت یک مجموعه از اجتماع آنها می‌نویسیم.

مثال : می‌خواهیم دیاگرام LALR مربوط به مثال صفحهٔ قبل را رسم نمائیم.

- وضعیت‌های سوم و ششم با هم ادغام شده و وضعیت 36 بدست خواهد آمد.
- وضعیت‌های چهارم و هفتم با هم ادغام شده و وضعیت 47 بدست خواهد آمد.
- وضعیت‌های هشتم و نهم با هم ادغام شده و وضعیت 89 بدست خواهد آمد.

پس خواهیم داشت :



	c	d	$\$$	S	C
S_0	S_{36}	S_{47}		1	2
S_1			Acc		
S_2					5
S_3	S_{36}	S_{47}			89
S_4	R_3	R_3			
S_5			R_1		
S_6					
S_7			R_3		
S_8	R_2	R_2			
S_9			R_2		

نکته :

- در صورتیکه گرامری CLR(1) باشد و پس از ادغام وضعیت‌های آن ، تداخلی در خانه‌ها نداشته باشیم (یعنی خانه‌ای با دو قانون نداشته باشیم) ، در اینصورت گرامر مورد نظر ، LALR(1) است.
- اگر گرامری CLR(1) نباشد حتماً LALR(1) هم نیست.
- ممکن است گرامری CLR(1) باشد ولی LALR(1) نباشد. در اینصورت جدولی داریم که تداخل $Reduce, Reduce$ خواهد داشت.
- تداخل $Shift, Reduce$ هیچ موقع پیش نخواهد آمد ؛ زیرا در صورتی این تداخل بوجود خواهد آمد که CLR(1) نباشد.

مثال : نشان دهید گرامر زیر ، LALR(1) نیست.

$$S \to aAd \quad S \to bBd \quad S \to aBe \quad S \to bAe \quad A \to c \quad B \to c$$

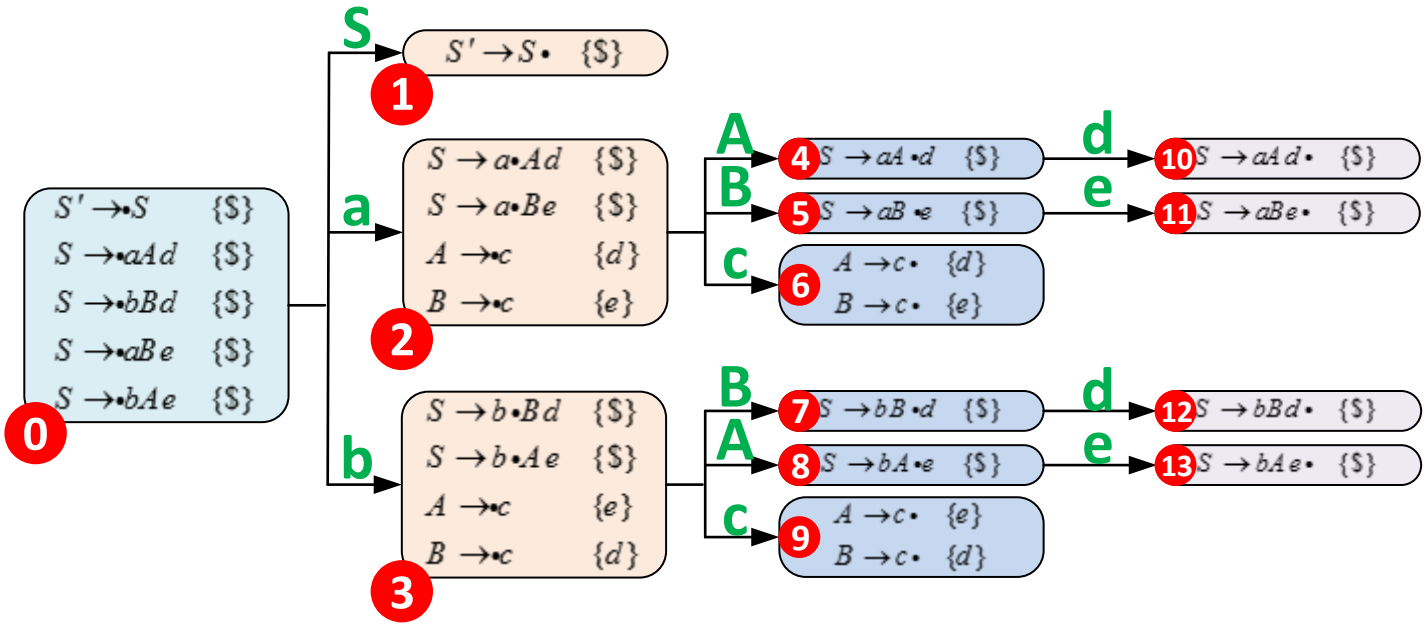
➔ در اولین گام ، یک قانون جدید به آن اضافه می‌کنیم ➔

$$S' \to S \quad S \to aAd \quad S \to bBd \quad S \to aBe \quad S \to bAe \quad A \to c \quad B \to c$$

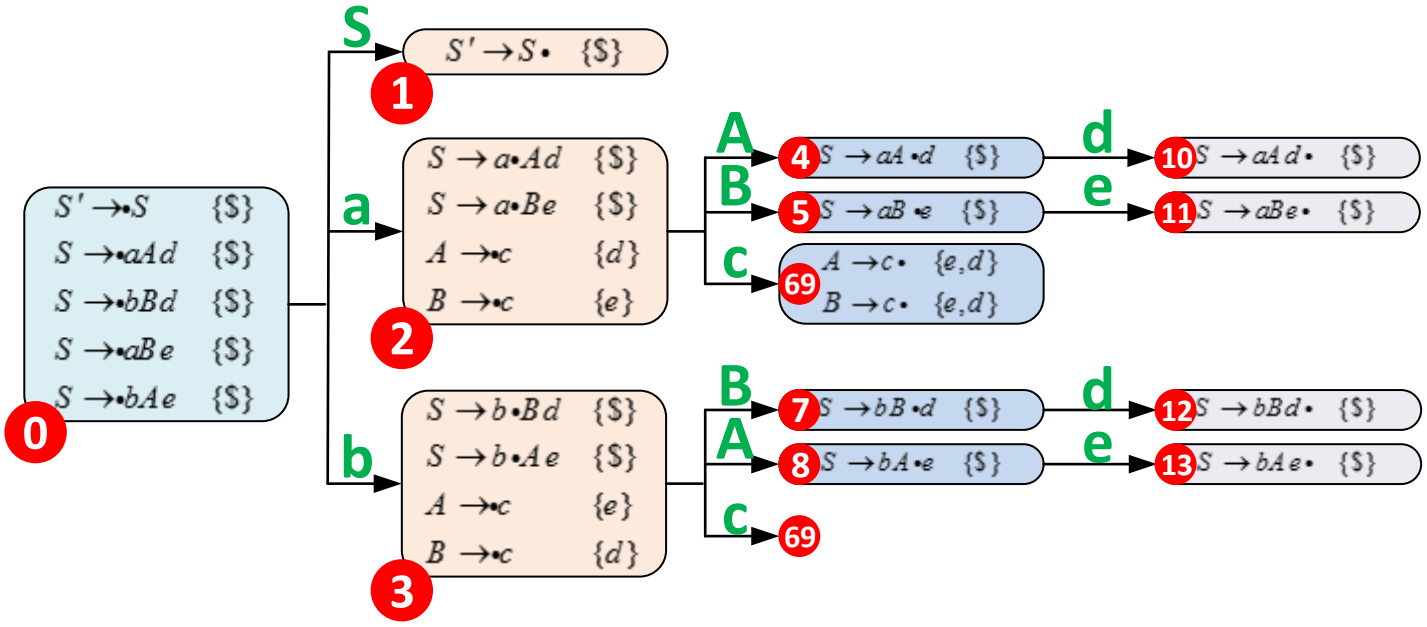
➔ در نهایت با شماره گذاری قوانین خواهیم داشت ➔

1. $S \to aAd$
2. $S \to bBd$
3. $S \to aBe$
4. $S \to bAe$
5. $A \to c$
6. $B \to c$

نمودار CLR آن بصورت زیر است :



و نمودار LALR آن بصورت زیر است :



جدول تجزیهٔ LALR نیز بصورت زیر خواهد بود :

	a	b	c	d	e	$	A	B	S
S_0	S_2	S_3							1
S_1						Acc			
S_2			S_{69}				4	5	
S_3			S_{69}				8	7	
S_4				S_{10}					
S_5					S_{11}				
S_6				~~$R_{5,6}$~~	~~$R_{5,6}$~~				
S_7				S_{12}					
S_8					S_{13}				
S_9									
S_{10}									
S_{11}									
S_{12}									
S_{13}									

مقایسهٔ جداول SLR ، CLR و LALR :

- تعداد وضعیت‌های جدول SLR و LALR یکسان است ولی جدول CLR از هر دو بیشتر است (در صورتی مساوی است که هیچ دو وضعیتی ادغام نشوند)
- تجزیه‌ای که به کمک جدول CLR انجام می‌شود ، خطاهای نحوی را سریعتر مشخص می‌کند. این نکته را در مثال زیر نشان می‌دهیم.

مثال : نشان دهید رشتهٔ ورودی $W = ccd$ در گرامر زیر ، در صورتیکه از جدول CLR به جای LALAR استفاده کند ، سریعتر خطا را مشخص خواهد کرد ؟

گرامر آن در زیر داده شده و برای جداول نیز می‌توانید به صفحات قبل مراجعه نمائید :

$S \to CC$

$S \to cC \mid d$

« با استفاده از جدول LALR »

Stack	Input	Action
0	$ccd\,\$$	$Shift\ 36$
$0c36$	$cd\,\$$	$Shift\ 36$
$0c36c36$	$d\,\$$	$Shift\ 47$
$0c36c36d47$	$\$$	$Reduce\ 3 \quad C \to d$
$0c36c36C89$	$\$$	$Reduce\ 2 \quad C \to cC$
$0c36C89$	$\$$	$Reduce\ 2 \quad C \to cC$
$0C2$	$\$$	$Error$

« با استفاده از جدول CLR »

Stack	Input	Action
0	$ccd\,\$$	$Shift\ 3$
$0c3$	$cd\,\$$	$Shift\ 3$
$0c3c3$	$d\,\$$	$Shift\ 4$
$0c3c3d4$	$\$$	$Error$

همانطور که مشخص است ، با استفاده از CLR در چهار مرحله و با استفاده از LALR در هفت مرحله ، خطا شناسایی شد.

خلاصه‌ای از فصل تحلیل نحوی :

شامل دو نوع تجزیه :

۱. بالا به پائین : که همان اشتقاق بود. با استفاده از روش تجزیهٔ پیش‌نگر که برگشت به عقب نداشت انجام می‌شد. بر روی گرامرهای LL(1) قابل استفاده بود که این گرامرها دارای سه شرط بودند :

a. رفع ابهام شده b. حذف چپ گردی c. حذف فاکتور چپ

جدول تجزیه در این مدل ، با استفاده از دو تابع First و Fallow و اصلاح خطا به دو روش Panic Mode و Phraze Level انجام می‌شد.

۲. روش پائین به بالا یا انتقال کاهشی که خود به سه دسته تقسیم می‌شد : (اصلاح خطا بوسیلهٔ مجموعهٔ هماهنگی صورت می‌گرفت)

a. روش OP : برای گرامرهای عمل‌گرا. جدول تجزیه با استفاده از دو تابع FirstTerm و LastTerm ساخته می‌شد که در سطر و ستون آن ، پایانی‌ها به همراه $ قرار می‌گرفت. داخل این جدول نیز از علامت‌های $\doteq, <\bullet, \bullet>$ استفاده می‌شد.

b. روش SP : برای گرامرهایی بود که قانون λ نداشت. جدول روابط تقدمی در این روش ، با استفاده از دو تابع Head و Tail ساخته می‌شد که در سطر و ستون آن ، علاوه بر پایانی‌ها و $ ، متغیرها نیز قرار داشتند.

c. روش‌های LR : تجزیه در این روش بوسیلهٔ وضعیت‌ها با سه روش SLR ، CLR و LALR انجام می‌شد. در این روش ، جدولی ساخته می‌شد که حاوی سه بخش State ، Action و Goto بودند.

« فصل چهارم : فاز تحلیل معنایی و تولید کد میانی »

این تحلیل در دو نوع موجود است (یا در دو زمان انجام می‌شود) که عبارت است از :

۱. تحلیـل معنایی پویا : (در زمان اجرا)

زمانی است که در یک تکه کد ، تحلیل معنایی آن در زمان کامپایل ، امکان پذیر نباشد. برای مثال در کد روبرو ، مقدار i بعد از اجرا مشخص خواهد شد. در اینصورت ، تحلیل معنایی ، بایستی در زمان اجرا انجام شود. برای اینکار ، کامپایلر کدهایی را به برنامه اضافه می‌کند که بتواند تحلیل معنایی مورد نظر را در زمان اجرا انجام دهد.

```
int i;
int A[10];
cin >> i;
A[i] = 5;
```

۲. تحلیل معنایی ایستا : (در زمان کامپایل) : این تحلیل معنایی ، شامل چند نوع به شرح زیر است :

a. بررسی نوع یا Type Checking : به کمک اطلاعاتی که در جدول نمادها قرار دارد ، یکسان بودن نوع متغیرها را بررسی می‌کند.

b. تست یکتایی یا Uniqueness Check : نامی که برای یک متغیر تعریف می‌کنیم ، برای یک آرایه یا یک تابع در همان محدوده ، قابل استفادهٔ مجدد نخواهد بود.

c. بررسی ساختارهای تو در تو و ورودی توابع : منظور از بررسی ساختارهای تو در تو ، زبان‌هایی مانند زبان Ada است. در این زبان ، حلقه‌ها دارای یک نامِ شروع می‌باشند که پایان آنها نیز با همین نام تعیین می‌شود (For 1, 2, ..). تست درستی این همنامی ، بر عهدهٔ تحلیل معنایی است. تگ‌ها در وب نیز مثال دیگری از این ساختارهای تو در تو هستند. ساختارهای تو در تو از لحاظ ظاهری ، بوسیلهٔ پرانتزها و در تحلیل نحوی بررسی می‌شوند. تحلیل معنایی ، تعداد ورودی توابع را نیز بررسی می‌کند. اگر این تعداد یکسان نباشد ، پیغام خطا می‌دهد.

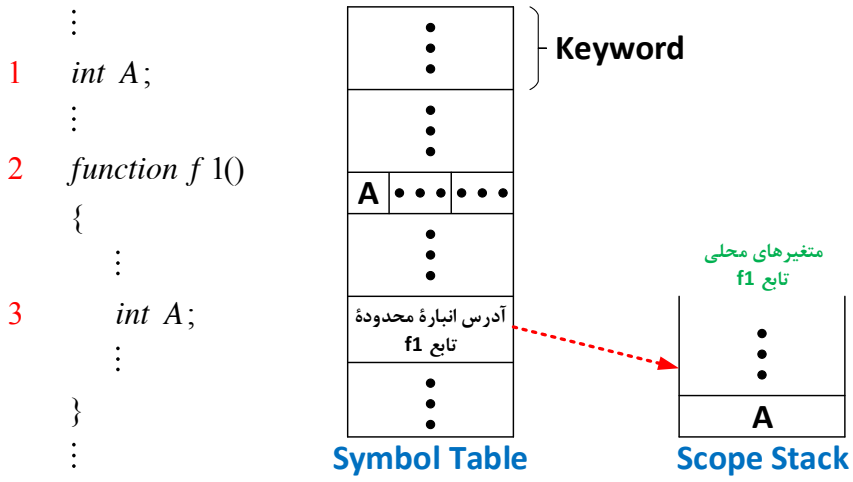
d. بررسی مسیر انتقال کنترل : برای مثال ، موقعیت دستور Break را بررسی می‌کند. اگر این دستور خارج از یک حلقه باشد ، یک خطای معنایی رخ داده است.

انبارهٔ محدوده یا Scope Stack :

حافظه‌هایی که تا اینجا برای کامپایلر به آن رسیدیم عبارتند از جدول نمادها یا Symbol Table و Syntax Stack. انبارهٔ محدوده ، حافظهٔ بعدی کامپایلر است که ساختاری به شرح زیر دارد :

به محض رسیدن به دستور اول ، A توسط تحلیل نحوی داخل جدول نمادها قرار می‌گیرد. همچنین به محض رسیدن به دستور دوم ، یک آدرس محدوده ایجاد شده و آدرس آن در جدول نمادها درج می‌شود. در واقع ، به تعداد توابع ، انبارهٔ محدوده داریم که آدرس آنها در جدول نمادها ثبت می‌شود.

از ایـن انبـاره ، برای فراخوانی‌های بازگشتی توابع استفـاده می‌شود. با رسیـدن به آخر تابع ، یکی یکی POP می‌شوند.

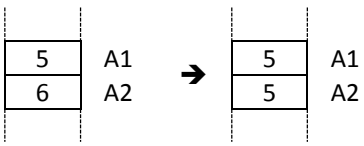


تولید کد میانی یا کدهای سه آدرسه :

یک چهارتایی مرتب است که معمولاً از یک عملیات یا عملگر و حداکثر سه آدرس تشکیل شده است. توجه داشته باشید که حتی در صورت کم شدن آدرس‌ها ، درج کاما در این چهارتایی ، کم یا حذف نخواهد شد. شکل کلی آن بصورت زیر است :

(آدرس سوم ، آدرس دوم ، آدرس اول ، عملگر)

مثال :



محتوای خانهٔ A1 را در خانهٔ A2 قرار بده (:= , A1 , A2 ,) ⇒

پرش شرطی : اگر محتوای خانهٔ A1 ، مقدار آن False باشد ، به A2 پرش کن (JPF , A1 , A2 ,) ⇒

پرش غیر شرطی : به خانهٔ A1 پرش کن (JP , A1 , ,) ⇒

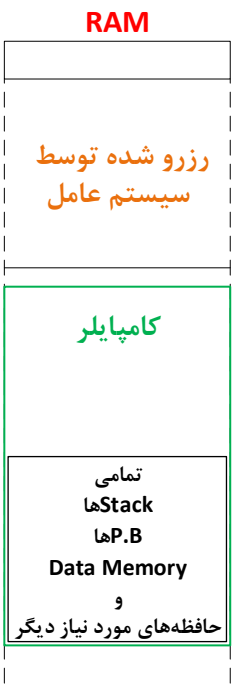
تولید کد میانی نیز به دو روش انجام می‌شود :

۱. پائین به بالا ۲. بالا به پائین

تولید کد میانی به روش بالا به پائین :

زمانیکه تحلیل نحوی به روش بالا به پائین انجام شود ، تولید کد میانی نیز به همین روش انجام می‌گیرد. برای بررسی آن نیاز به یکسری تعاریف پایه‌ای داریم که به آنها می‌پردازیم :

- Semantic Stack (S.S) : که برای تولید کدهای سه آدرسه از آن استفاده شده و داخل آن عملاً تعداد آدرس از خانه‌های مختلف قرار می‌گیرد.
- Action Symbols (A.S) یا نمادها یا علائم کنش : عبارتند از یکسری علامت که داخل گرامر زبان اضافه می‌شوند. وظیفهٔ این نمادها ، کمک برای تولید کدهای سه آدرسه است.
- Semantic Action (S.A) یا معنای کنش‌ها : معادل هر نماد کنش ، یک عملیات برای ساخت کد میانی انجام می‌شود که به این عملیات‌ها ، S.A گفته می‌شود. در واقع یک S.A ، مفهوم آن عملیات را نشان می‌دهدکه به آنها روال‌های مفهومی نیز گفته می‌شود.
- Program Block (P.B) : بخشی از حافظهٔ زمان اجراست که بصورت ایستا ، معمولاً برای کامپایلرها تخصیص داده می‌شود. محتوای آن ، کدهای سه آدرسهٔ تولید شده می‌باشد که هر کد در یک خانه از P.B قرار می‌گیرد. در واقع مشابه آرایه‌ای است که در هر خانهٔ آن ، یک کد سه آدرسه قرار دارد که برای دسترسی به این کد سه آدرسه نیز از $PB[i]$ استفاده می‌شود.
- حافظهٔ داده‌ها یا Data Memory : محدوده‌ای از حافظهٔ اصلی است که برای تعریف متغیرهای برنامه ، معمولاً بصورت ایستا برای کامپایلر تعریف می‌شود. توجه داشته باشید که حافظه‌ها در کامپایلرهای امروزی بصورت لیست پیوندی هستند.



در اینجا چند قرارداد با هم خواهیم بست :

- P.Bها درخانه‌های 1 تا 99 قرار خواهند گرفت
- Data Memoryها درخانه‌های 100 تا 499 قرار خواهند گرفت
- خانهٔ 500 از حافظه هم ، موقت است (Temp)

تولید کد میانی مربوط به عبارات جبری و دستور انتساب :

در گرامر فوق ، نمادهای قرمز رنگ که با علامت Sharp شروع می‌شوند ، همان Action Symbolها هستند. جای این A.Sها مهم است و بستگی به عملیاتی که انجام می‌دهند ، دارد.

مهمترین کار در تولید کد میانی ، جای همین A.Sها است. معادل هر کدام از A.Sها ، یک عملیات تعریف می‌شود که در واقع همان S.A ها می‌باشند. با این عملیات ، کد سه آدرسهٔ معادل دستور ساخته شده و درون P.B قرار می‌گیرد.

برای ایجاد کدهای سه آدرسه از پشتهٔ معنایی یا Semantic Stack یا به اختصار S.S که آدرس متغیرها را در خود نگهداری می‌کند ، استفاده می‌شود. در واقع ، هر جایی که عملیاتی انجام می‌شود ، یک Action Symbol قرار می‌گیرد.

1. $S \rightarrow L := E \quad \#assign$
2. $E \rightarrow TE'$
3. $E' \rightarrow \lambda$
4. $E' \rightarrow +T \quad \#add \quad E'$
5. $T \rightarrow FT'$
6. $T' \rightarrow \lambda$
7. $T' \rightarrow *F \quad \#mul \quad T'$
8. $F \rightarrow (E)$
9. $F \rightarrow \#pid \quad id$
10. $L \rightarrow \#pid \quad id$

عملیات pid Semantic Action یا pid :

که در واقع ، منظور $Push \quad id$ است.

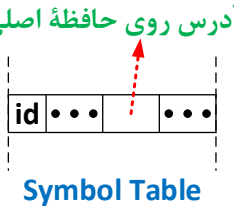
با رسیدن به خط دوم ، آدرس واقعی id مورد نظر ، از جدول نمادها پیدا شده و درون p قرار داده می‌شود.

با رسیدن به خط سوم ، آدرس یافت شده در خط قبل ، داخل پشتهٔ معنایی یا Semantic Stack قرار داده می‌شود.

در واقع یکی دیگر از چیزهایی که که در جدول نمادها و در جلوی idها قرار می‌گیرد ، آدرس اصلی آنها روی حافظهٔ اصلی است. این آدرس ، طبق قراردادی که در بالا هم به آن اشاره شد ، از خانهٔ شمارهٔ 100 الی 499 است که بر روی Data Memory قرار دارد.

در این عملیات ، هیچ کد سه آدرسه‌ای تولید نخواهد شد ؛ تنها یک خانه از حافظه ، به روی Stack انتقال می‌یابد.

```
1   #pid : Begin
2       p ← find Addr(input)
3       Push(p)
4   End
```



عملیات add یا جمع :

برای جمع ، نیاز به یک خانهٔ موقت داریم تا نتیجه داخل آن قرار داده شود. این عمل در خط دوم از کد روبرو انجام می‌شود.
در این عملیات ، بایستی کد سه آدرسهٔ مربوط به جمع تولید شود. لذا آدرس دو خانه را از پشتهٔ معنایی یا S.S برداشته ، با هم جمع کرده و نتیجه را داخل خانهٔ خالی یافت شده در مرحلهٔ قبل ، قرار می‌دهد. در نهایت ، کد سه آدرسهٔ تولید شده ، درون P.B قرار می‌گیرد.
در خط پنجم ، دو خانهٔ بالای پشته را حذف می‌کند.
در خط ششم نیز ، نتیجهٔ جمع را به جای دو عملوند قبلی ، داخل پشته قرار می‌دهد.

```
1  #add : Begin
2      t ← gettemp
3      PB[i] ← (+, SS(top), SS(top − 1), t)
4      i ← i + 1
5      Pop(2)
6      Push(t)
7  End
```

عملیات mul یا ضرب :

نسبت به عملیات جمع ، تنها این کدهای آبی رنگ هستند که تغییر کرده‌اند.
بقیهٔ حالات و توضیحات ، همانند عملیات جمع هستند.

```
1  #mul : Begin
2      t ← gettemp
3      PB[i] ← (*, SS(top), SS(top − 1), t)
4      i ← i + 1
5      Pop(2)
6      Push(t)
7  End
```

عملیات assign یا انتساب :

در این عملیات ، Push وجود ندارد ؛ زیرا چیزی تولید نمی‌شود.

```
1  #assign : Begin
2      PB[i] ← (:=, SS(top), SS(top − 1), )
3      i ← i + 1
4      Pop(2)
5  End
```

مثال : برای عبارت $a := b + c * d$ با توجه به جدول تجزیهٔ زیر ، P.B مورد نظر را بدست آورید ؟

1. $S \rightarrow L := E \quad \#assign$
2. $E \rightarrow TE'$
3. $E' \rightarrow \lambda$
4. $E' \rightarrow +T \quad \#add \quad E'$
5. $T \rightarrow FT'$
6. $T' \rightarrow \lambda$
7. $T' \rightarrow *F \quad \#mul \quad T'$
8. $F \rightarrow (E)$
9. $F \rightarrow \#pid \quad id$
10. $L \rightarrow \#pid \quad id$

A \ a	id	+	*	(	)	:=	$
S	1						
L	10						
E	2			2			
E'		4			3		3
T	5			5			
T'		6	7		6		6
F	9			8			

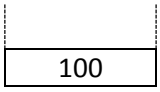
شکل کلی عبارت بصورت $id_1 := id_2 + id_3 * id_4$ است.

تجزیهٔ عبارت مورد نظر ، بصورت زیر خواهد بود :

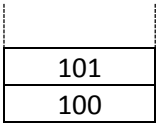
#	Stack	Input	Description
1	$S	$id_1 := id_2 + id_3 * id_4$	
2	$ #assign E := L	$id_1 := id_2 + id_3 * id_4$	
3	$ #assign E := id #pid	$id_1 := id_2 + id_3 * id_4$	Push id
4	$ #assign E :=	$:= id_2 + id_3 * id_4$	Delete :=
5	$ #assign E	$id_2 + id_3 * id_4$	
6	$ #assign E T	$id_2 + id_3 * id_4$	
7	$ #assign E T F	$id_2 + id_3 * id_4$	
8	$ #assign E T id #pid	$id_2 + id_3 * id_4$	Push id
9	$ #assign E T'	$+id_3 * id_4$	
10	$ #assign E'	$+id_3 * id_4$	
11	$ #assign E' #add T +	$+id_3 * id_4$	Delete +
12	$ #assign E' #add T	$id_3 * id_4$	
13	$ #assign E' #add T F	$id_3 * id_4$	
14	$ #assign E' #add T id #pid	$id_3 * id_4$	Push id
15	$ #assign E' #add T'	$*id_4$	
16	$ #assign E' #add T' #mul F *	$*id_4$	Delete *
17	$ #assign E' #add T' #mul F	id_4	
18	$ #assign E' #add T' #mul id #pid	id_4	Push id
19	$ #assign E' #add T' #mul	$	
20	Multiplication		
21	Summation		
22	Assignation		

« نمایش Scope Stack پس از اجرا شدن دستورات مختلف »

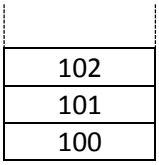
« پس از دستور سوم »



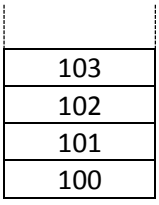
« پس از دستور هشتم »



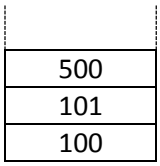
« پس از دستور چهاردهم »



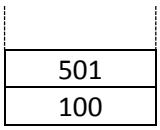
« پس از دستور هجدهم »



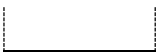
« پس از دستور بیستم »



« پس از دستور بیست و یکم »



« پس از دستور بیست و دوم »



« نمایش P.B پس از اجرا شدن دستورات مختلف »

« دستور بیستم »

0	*, 103, 102, 500

« دستور بیست و یکم »

1	+, 500, 101, 501
0	*, 103, 102, 500

« دستور بیست و دوم »

2	:=, 501, 100,
1	+, 500, 101, 501
0	*, 103, 102, 500