به نام خدا

۱ ۹۱/۱۰/۲۱

جزوه اصول طراحی کامپایلر — فصل ۱ مقدمه ای بر کامپایلر

* ① زبان ماشین: زبانی که داده ها و دستورالعمل ها به صورت کدهای باینری نمایش داده میشوند. (برنامه نویسی مشکل و زمانبر)

* ② زبان اسمبلی: زبانی که به جای کدهای باینری از کلمات اختصاری استفاده می کند و خوانائی بیشتری نسبت به زبان ماشین دارد.

* اسمبلر: نرم افزار مترجمی است که زبان اسمبلی را به زبان ماشین ترجمه می کند

* ③ زبان های سطح بالا: زبان های سطح بالا به زبان محاوره ای نزدیکترند و ساختار و دستورات قدرتمندی نسبت به اسمبلی دارند. (C و پاسکال)

* برنامه های سطح بالا توسط کامپایلر به زبان ماشین ترجمه می شوند

* تفاوت اسمبلر و کامپایلر: اسمبلر هر دستور اسمبلی را فقط به یک دستور ماشین ترجمه می کند درحالیکه کامپایلر هر دستور زبان سطح بالا را به چندین دستور زبان ماشین ممکن است ترجمه کند به عبارت دیگر اسمبلی یک به یک و کامپایلر یک به چند است.

* روش های ترجمه و اجرای برنامه های سطح بالا: ۱- استفاده از مفسر ۲- استفاده از کامپایلر

* ① مفسر: دستورالعمل های برنامه یک به یک توسط مفسر که یک نرم افزار است خوانده میشود و اجرا می گردد. در این روش برنامه مبدأ به زبان ماشین ترجمه نمی گردد و در نتیجه فایل جداگانه ای تولید نمی شود و اجرای برنامه سرعت خوبی دارد (مفسد برای برنامه در حال ساخت)

* مزایای استفاده از مفسر: ۱- سهولت اشکال زدائی و کاهش زمان ساخت برنامه ۲- قابلیت انعطاف بالا ۳- پیاده سازی آسان

۴- قابلیت حمل بالا * معایب استفاده از مفسر: ۱- تکرار تفسیر ۲- سرعت اجرای پائین ۳- نیاز به مفسر ۴- دسترسی به کد منبع

* ② کامپایلر: برنامه به وسیله کامپایلر به زبان ماشین ترجمه میشود (برنامه مبدأ باید صحیح باشد)

* مزایای استفاده از کامپایلر: ۱) سرعت اجرای بالا ۲) اجرای مستقل برنامه از کامپایلر ۳) حفاظت از کد منبع ۴- عدم تکرار کامپایل

* معایب کامپایلر: ۱) زمانبر بودن اشکال زدائی ۲) قابلیت حمل پائین ۳) پیچیدگی پیاده سازی ۴) نیاز به هر کامپایلر برای هر ماشین

* بهترین وضعیت استفاده از هر دو روش یعنی در هنگام تولید از مفسر و هنگام تولید نهائی از کامپایلر استفاده شود مثل Visual basic

* زبان C از کامپایل و زبان لیسپ از تفسیر استفاده می کنند. * زبانهای script توسط مرورگر تفسیر می شوند

* زبان جاوا دارای یک کامپایلر است که به جای ترجمه کد جاوا به زبان ماشین، برنامه را به یک ماشین منطقی و مجازی به نام JVM ترجمه می کند

* JVM برنامه کامپایل شده را به روش تفسیر اجرا می کند * C# مثل جاوا بوده و CLR نقش ماشین مجازی را دارا می باشد

* مشکل زبانهائی مثل C# و جاوا سرعت اجرای پائین برنامه است زیرا یک لایه (ماشین مجازی) بین سخت افزار و برنامه قرار دارد

برای حل این مشکل این زبانها به گونه ای طراحی شده اند که بتوان آنها را به زبان ماشین واقعی نیز تبدیل و ترجمه نمود. برنامه تبدیل شده

اجرا به زبان ماشین ترجمه شده و از اجرای بعد مورد استفاده قرار می گیرد.

کامپایلر – فصل ۱ ص ۲

* زبان پیاده سازی کامپایلر: کامپایلر خود نیز توسط یک زبان برنامه نویسی به نام زبان پیاده ساز نوشته میشود که این زبان نیز توسط کامپایلر دیگری دارد و این روند ادامه می یابد. اولین کامپایلر به زبان اسمبلی نوشته شده و اسمبلر آن را به زبان ماشین ترجمه می کند.

* جلوبندی و عقب بندی کامپایلر: برای حل مشکل تعداد زیاد کامپایلرها از این روش استفاده می گردد. راه حل استفاده از یک زبان میانی است که به جای mk کامپایلر فقط m+k کامپایلر نیاز است. بخشی از کامپایلر که وظیفه ترجمه برنامه مبدأ به برنامه زبان میانی را دارد جلوبندی و بخش دیگر که وظیفه ترجمه برنامه زبان میانی به زبان مقصد را دارد عقب بندی کامپایلر نام دارد.

* جلوبندی کامپایلر: زبان مبدأ و عقب بندی کامپایلر: زبان مقصد وابسته است * مزایای جلوبندی و عقب بندی کامپایلر:

۱) سادگی طراحی ۲) استقلال جلوبندی از زبان مقصد ۳) استقلال عقب بندی از زبان مبدأ ۴) کاهش پیچیدگی ۵) استفاده مجدد ۶) سرعت

* IDE: محیطی برای برنامه نویسی که همه امکانات لازم برای نوشتن برنامه، خطایابی و تولید کد نهایی را در اختیار کاربر قرار می دهد.

* پیش پردازنده: وظیفه انجام برخی تبدیلات اولیه را بروی برنامه ورودی بر عهده دارد، یک کامپایلر سطح پایین است که برخی ترجمه ها را انجام میدهد:

* وظایف پیش پردازنده: ۱) پردازش ماکروها ۲) الحاق فایل ها و توابع کتابخانه ای ۳) تعمیم زبان

* کامپایلر برنامه خروجی پیش پردازنده را به زبان اسمبلی ترجمه می کند، این بخش مهمترین و بیشترین وظایف در تبدیل برنامه مبدأ به یک برنامه اجرایی را بر عهده دارد. * بارکننده و ویرایشگر پیوند: انتقال برنامه اجرایی به درون حافظه به منظور اجرا

* فازهای کامپایلر: ۱) تحلیلگر لغوی ۲) تحلیلگر نحوی ۳) تحلیلگر معنایی ۳-۱) بررسی هماهنگی پارامترها ۳-۲) بررسی و کنترل نوع
۳-۳) تبدیل نوع ، ۳-۴) بررسی و تعریف دوباره متغیر ۴) تولید کننده کد میانی ۵) بهینه کننده کد میانی ۶) تولید کننده کد

* مدل تجزیه و ترکیب: کامپایلر شامل ۲ مرحله است. در مرحله اول برنامه، اجزای تشکیل دهنده آن ترجمه میشود و در طی این مرحله فازهای تحلیل نحوی، لغوی و معنایی صورت پذیرفته و خطاها مشخص میشود همچنین تولید کد میانی در این مرحله است که کل این اجزا جلوبندی کامپایلر هستند. پس از اطمینان از صحت برنامه مبدأ، برنامه تجزیه شده برای ایجاد برنامه مقصد ترکیب می گردد. فازهای بعدی عملیات ترکیب را انجام داده و عقب بندی کامپایلر را تشکیل می دهند.

* مزایای تقسیم کامپایلر به فازها: ۱) سادگی طراحی ۲) افزایش کارایی ۳) افزایش قابلیت حمل ۴) افزایش قابلیت انعطاف پذیری

* مفسر که فاز تحلیل لغوی، نحوی، معنایی و تولید کد میانی را انجام می دهند و سپس به جای ترجمه به زبان ماشین کد میانی را اجرا می کنند.

* نرم افزارهایی که دستورات SQL را اجرا می کنند فازهای تحلیل لغوی و نحوی را انجام می دهند.

* ویژگی کامپایلر خوب: ۱) تولید کد صحیح ۲) پیروی از مشخصات زبان مبدأ ۳) پردازش برنامه های بزرگ ۴) سرعت خوب ۵) اندازه مناسب
۶) پیغام خطای واضح ۷) قابلیت حمل ۸) بهینه سازی

نحوه کامپایلر فصل ۲ تحلیلگر لغوی ص ۳ * اولین فاز جلوبندی کامپایلر تحلیلگر لغوی می باشد.

* تحلیلگر لغوی تنها فازی است که مستقیما با برنامه مبدأ در ارتباط است. این فاز برنامه ورودی را بررسی نموده، لغات و نوع آنها را تشخیص داده و برای تحلیلگر نحوی ارسال می کند. به خواندن برنامه ورودی و استخراج لغات توسط تحلیلگر لغوی پویش گر گویند.

* انواع لغات برنامه: ۱) کلمات کلیدی مثل main در C و program در پاسکال * زبان PL/1 اجازه استفاده از کلمه کلیدی را می دهد ولی پاسکال و C اجازه استفاده از کلمه کلیدی به عنوان شناسه را نمی دهند در این زبانها به کلمات کلیدی کلمات رزرو شده می گویند

۲) سمبل ها: به منظور نشانه گذاری خاص در زبان مبدأ به کار می روند (؛ در C و (؛) در پاسکال

۳) عملگرها: به منظور نشان دادن عملیات خاص در برنامه مثل (/ ، * ، ? ، + ، = ! ، < و >) در C و (+ ، < ، >) در پاسکال

۴) شناسه ها: نام هایی که خود کلمات کلیدی زبان مبدأ نیستند و توسط برنامه نویس تعریف می گردند مثل: نام متغیر، نام آرایه، نام کلاسها

۵) ثوابت: مقادیر عددی، رشته ای، کاراکتری و یا منطقی که برنامه نویس در برنامه استفاده می کند.

۶) توضیحات: قسمتی از برنامه است که توسط کامپایلر در نظر گرفته نمی شود

۷) فضای خالی: تأثیر فضای خالی در برنامه شکلی زبان مبدأ را در زبان C و پاسکال از فضای خالی به منظور خوانائی برنامه استفاده میشود

* تحلیلگر لغوی ساختار تک تک لغات را جدا از هم بررسی می کند و خطای استفاده از متغیری که تعریف نشده است را کشف نمی کند.

* اگر لغتی با ساختارهای لغوی مورد انتظار تحلیلگر لغوی مطابق نباشد آنرا خطا در نظر می گیرد و این عملگر برخورد با خطا شبیه به نحو تحلیلگر لغوی دارد. در کل تحلیلگر لغوی به چند شکل با خطا برخورد می کند که عبارتند از:

۱) توقف ۲) پوشش خطا: در این روش تحلیلگر پس از برخورد با خطا متوقف نمی شود (حذف کاراکترهای ورودی تا زمان یافتن لغت معتبر)

* نشانه *: فاز بعد از تحلیلگر لغوی، تحلیلگر نحوی هست که باید علاوه بر وجود لغات، نوع هر لغت را نیز در اختیار داشته باشد. نشانه علامتی است که تحلیلگر لغوی و تحلیلگر نحوی برای مشخص کردن یک نوع خاص از لغات با هم توافق کرده اند.

* تحلیلگر لغوی جریانی از کاراکترها را بعنوان ورودی گرفته و جریانی از نشانه ها را بعنوان خروجی برای تحلیلگر نحوی ارسال می کند

* روش ارسال نشانه ها: ۱) استفاده از فایل واسط ۲) ارتباط مستقیم (این روش سریعتر است و اکثر کامپایلرها از آن استفاده می کنند)

* نحوه تخصیص نشانه ها به انواع لغات: ۱) کلمات کلیدی: برای هر کلمه کلیدی یک نشانه مجزا در نظر گرفته میشود

۲) علائم: می توان برای هر علامت یک نشانه در نظر گرفت اگر کاراکتری باشد می توان از کد اسکی به عنوان نشانه استفاده کرد

۳) عملگرها: می توان برای هر عملگر یک نشانه در نظر گرفت و یا یک نشانه برای چند عملگر مثل عملگرهای مقایسه ای در نظر گرفت.

۴) شناسه ها: تحلیلگر لغوی برای هر شناسه فقط یک نشانه در نظر می گیرد و به دلیل استفاده در فازهای بعدی از جدول نمادها استفاده میشود

* ادامه نشانه ها

۵) ثوابت: الگوی شناسه مطابق نوع ثابت است، هر ثابت اختصاص داده میشود علاوه بر آن مقدار ثابت نیز به تحلیلگر نحوی ارسال میشود.

۶) فضای خالی و توضیحات: چون تاثیری ندارند حذف میشوند و نشانه ای برای این دو مورد در نظر گرفته نمی شود.

* جدول نماد: با توجه به تخصیص یک نشانه برای چند لغت، تحلیلگر لغوی از جدول نماد برای هر نشانه یک رکورد در نظر گرفته و آدرس آن را برای تحلیلگر نحوی ارسال می کند این موارد شامل: متغیرها، نام زیر برنامه ها، آرایه ها، برچسب ها که خصوصیات آنها از قبیل نوع و آدرس ذخیره میشود

* پیاده سازی جدول نماد: ۱) لیست پیوندی ۲) جدول درهم ساز

* لیست پیوندی ساده ترین روش و زمان جستجو در آن خطی است برای افزایش کارایی از ساختار درختی نیز می توان استفاده نمود.

* جدول درهم ساز: مهمترین بخش درهم سازی تابع درهم ساز است که رشته ای را گرفته و عدد صحیحی بین 0 تا 1-m بازمی گرداند که این عدد صحیح به عنوان اندیس یا آدرس یک عنصری جدول درهم ساز استفاده میشود، اگر دو رشته با یک عدد درهم سازی شود برخورد رخ داده است.

* الگو: عبارتی هستند که ساختار مشترک مجموعه ای از لغات را بیان می کنند مثل doc.* یعنی همه فایل هایی که پسوند doc دارند

* زبانها: ۱) الفبای زبان: مجموعه ای ناتهی و متناهی از نمادهای ساده و غیر قابل تجزیه گویند و با Σ نمایش می دهند.

۲) رشته: یک دنباله متناهی از نمادهای الفبا را گویند. ۳) زبان: مجموعه ای از رشته ها را زبان گویند.

* عملیات در زبانها: ۱) اجتماع: مجموعه ای از رشته که یا در زبان L_1 یا در زبان L_2 باشند را با $L_1 \cup L_2$ نمایش می دهیم و خاصیت جابجایی دارد

۲) الحاق: به صورت $L_1.L_2$ نمایش داده و مجموعه ای از رشته که از ترکیب رشته ای در زبان L_1 و رشته ای در زبان L_2 به دست آمده باشد

این عمل یعنی الحاق برخلاف اجتماع خاصیت جابجایی ندارد.

۳) ستاره: اگر L یک زبان باشد ستاره L را به صورت L^* نشان می دهیم، L^* یعنی الحاق صفر یا بیشتر L با خودش

* انواع زبانها: ۱) زبانهای باقاعده یا Regulal Languages، ۲) زبانهای مستقل از متن یا Context free languages

۳) زبانهای وابسته به متن یا Context Sensitiv languag ۴) زبانهای قابل شمارش بازگشتی یا Recursiv enum langueges

* زبان باقاعده زبانی است که بتوان آن را توسط یک عبارت باقاعده تولید کرد. $\rightarrow L(r_1) \cup L(r_2)$

* عبارات باقاعده: ۱) ε یک عبارت باقاعده است ۲) به ازای هر a عضو Σ، $r=a$ یک عبارت باقاعده است $\leftarrow \{a\}$

۳) اگر $r=a$ عبارت باقاعده باشد، r^* نیز یک عبارت باقاعده است $\leftarrow (L(r))^*$ ۴) اگر r_1, r_2 عبارت باقاعده باشند $r=r_1|r_2$ باقاعده است

۵) اگر r_1 و r_2 عبارات باقاعده باشند $r=r_1.r_2$ باقاعده است $\leftarrow L(r_1).L(r_2)$

خلاصه کامپایلر فصل ۲ ص۵ * انواع زبانها

* ماشین خودکار متناهی: تشخیص دهنده زبانهای باقاعده بوده و در دو نوع هستند: ۱) ماشین خودکار قطعی DFA ۲) ماشین خودکار غیرقطعی NFA

* ماشین خودکار قطعی با DFA به صورت $M=(Q,\Sigma,\delta,q_0,F)$ که نمایش داده میشود که شامل بخش‌های زیر است:

۱) Q: مجموعه‌ای متناهی از حالات است، ۲) Σ: مجموعه‌ای متناهی از نمادکه الفبای زبان است، ۳) q_0: یک حالت از حالات ($q_0 \in Q$) به عنوان حالت شروع در نظر گرفته میشود ۴) F: زیرمجموعه‌ای از حالات ($F \subseteq Q$) به عنوان حالت یا حالات پایانی در نظر گرفته میشود

۵) δ: یک تابع گذر است که زوج متشکل از هر حالت q ($q \in Q$) و، برای هر یک از نمادهای الفبا ($a \in \Sigma$) به چه حالتی میرسد.

* ماشین خودکار قطعی با NFA به صورت $M=(Q,\Sigma,\delta,q_0,F)$ نمایش داده میشود و مثل DFA است فقط در مورد پنجمین متفاوت است،

۵) δ: یک تابع گذر است که زوج متشکل از هر حالت q ($q \in Q$) و، برای هر یک از نمادهای الفبا ($a \in \Sigma$) به چه حالتهایی میرسد.

* رشته x توسط DFA و یا NFA پذیرفته میشود اگر مسیری از حالت شروع به حالت پایان وجود داشته باشد، طوری که برچسب آن x را تشکیل دهد.

* روش تولید برنامه‌ای که لغات را پیدا کند و بررسی کند آیا توسط عبارت باقاعده ۲ قابل تولید هست یا خیر روش کلی متعددی وجود دارد.

یکی از این روش‌ها تبدیل عبارت باقاعده به NFA و سپس تبدیل NFA به DFA، برنامه‌ای که برای تبدیل عبارت باقاعده به DFA نیز روش‌های زیادی وجود دارد یکی از این روش‌ها تبدیل عبارت باقاعده به NFA و سپس تبدیل به DFA هست و برای دیگر تبدیل مستقیم به DFA

* ایجاد NFA از عبارت باقاعده، روش تامپسون:

۱) عبارت باقاعده را در نظر می‌گیریم و از روی نمادگی اولیه تجزیه می‌کنیم. (مثال)

	عبارت باقاعده	نمادگی اولیه عبارت باقاعده			
①	$r=a\|b$	$r_1=a$	$r_2=b$		
②	$r=(a\|b)^*(ca)^*$	$r_1=a$	$r_2=b$	$r_3=c$	$r_4=a$
③	$r=(\epsilon\|b\|c)^*$	$r_1=\epsilon$	$r_2=b$	$r_3=c$	

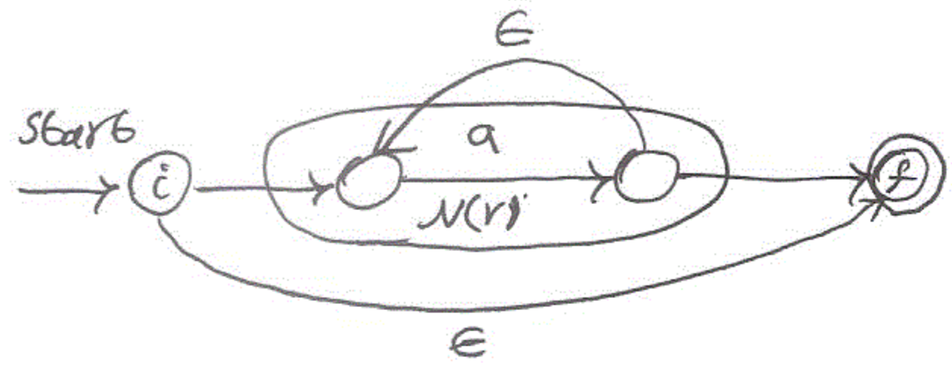
* برای هر یک از نمادگان یک NFA می‌سازیم که یک حالت شروع با نام i و یک حالت نهایی با نام f دارد

مثال NFA حالت ① را می‌سازیم در حالت ① با توجه به جدول بالا دو نماد وجود دارد پس نیاز به دو NFA هست

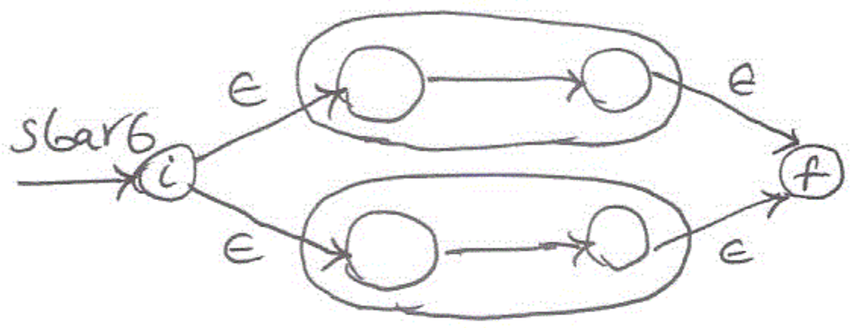
* الگوریتم ایجاد NFA به روش تامپسون

* طرحی به منظور سهولت کار در این جا، NFA های حالت همه را در جعبه می گذیم، تحلیل NFA بعهده خواننده واگذار میشود

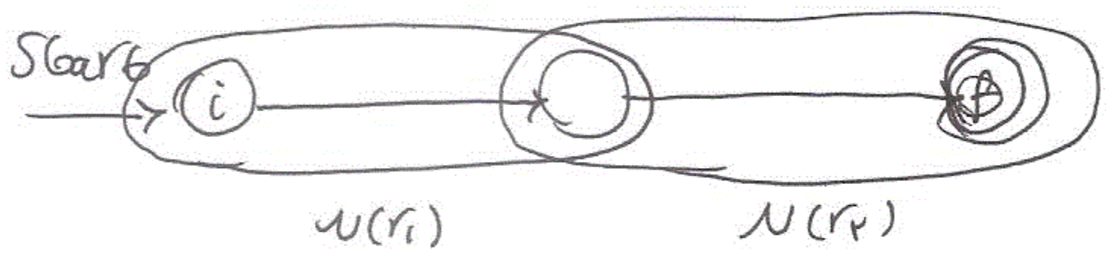
NFA ی a*



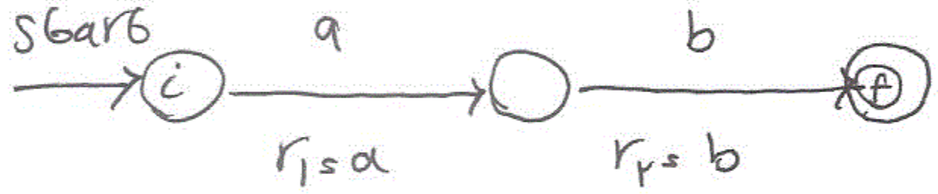
NFA ی r1|r2



NFA ی r1.r2

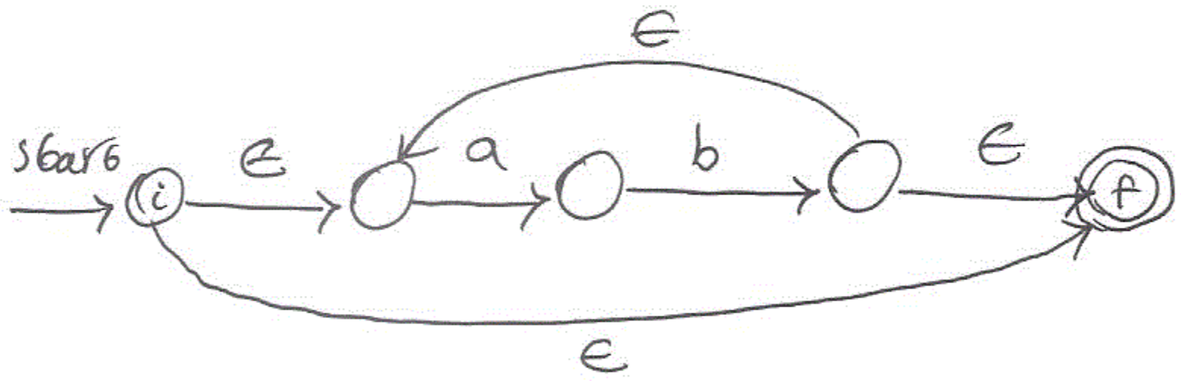


NFA ی s1 = ab



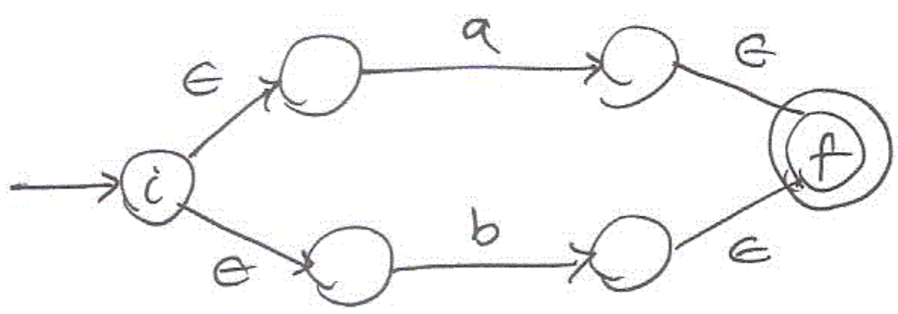
مثال ۴

NFA ی s1 = (ab)*



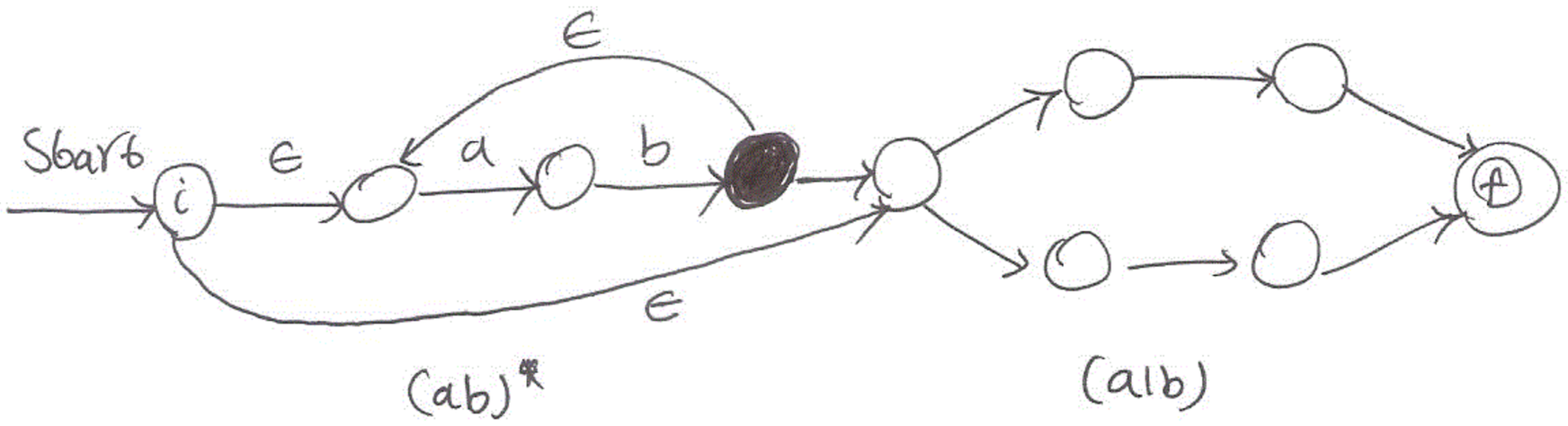
مثال ۴

NFA ی s2 = a|b



مثال ۴

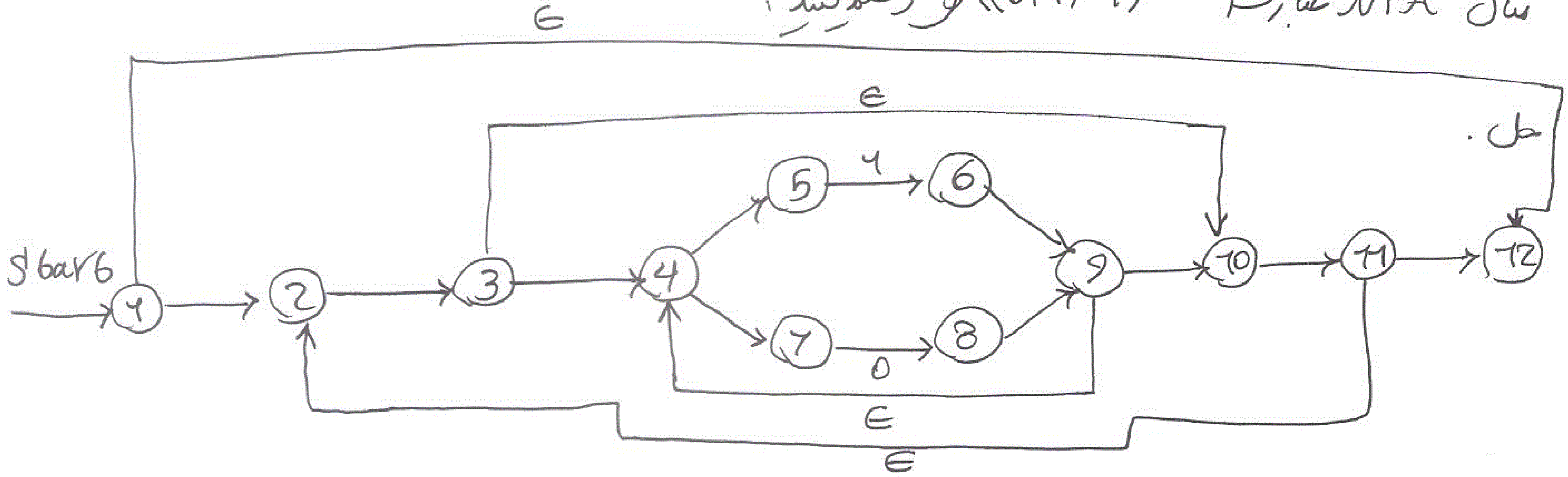
$s_3 = s_1.s_2 = (ab)^*.(a|b)$ ی NFA



جزوه کامپایلر فصل ۲ ص ۷ ادامه ایجاد NFA به روش تامپسون

مثال NFA عبارت $((0|1)^*1)^*$ را ترسیم کنید:

حل.



* ایجاد DFA از NFA: ابتدا نیازمند چند تعریف هستیم و سپس ادامه خواهیم داد.

* E_closure: مجموعه حالاتی از NFA که از حالت s و با تبدیل ε قابل دسترسی هستند.

* Move (T,a): مجموعه ای از حالات NFA است که از حالات درون مجموعه T با نماد a قابل دسترسی هستند.

* حال با توجه به E_closure و Move الگوریتمی ارائه می دهیم که NFA را به DFA تبدیل کند؟

۱) $E\_closure(s_0)$ را محاسبه می کنند (که حالت شروع است) و به مجموعه حاصل یک نام مثل A اختصاص داده و به جدولی به نام DTrans اضافه می کنند. این مجموعه حالت شروع DFA را تشکیل می دهد.

۲) یک حالت علامت نخورده درون DTrans یافته و کارهای زیر را انجام می دهیم، اگر حالت علامت نخورده در جدول نباشد پایان می یابد.

۳) حالت علامت نخورده را T نامیده و آن را علامت می زنند

۴) برای هر نماد الفبای زبان که آن را a می نامیم مراحل زیر را تکرار می کنند

۴-۱) مجموعه E_closure(Move(T,a)) را محاسبه می کنند و این مجموعه را (U) می نامند اگر این مجموعه با مجموعه های قبلی که در DTrans موجود است تفاوت داشت در آن یک نام جدید به آن داده و آن را به جدول اضافه می کنند

۴-۲) حالت بعدی T به ازای ورودی a است در نتیجه در DTrans تحت مورد را اضافه می کنند ← DTrans[T,a]=U

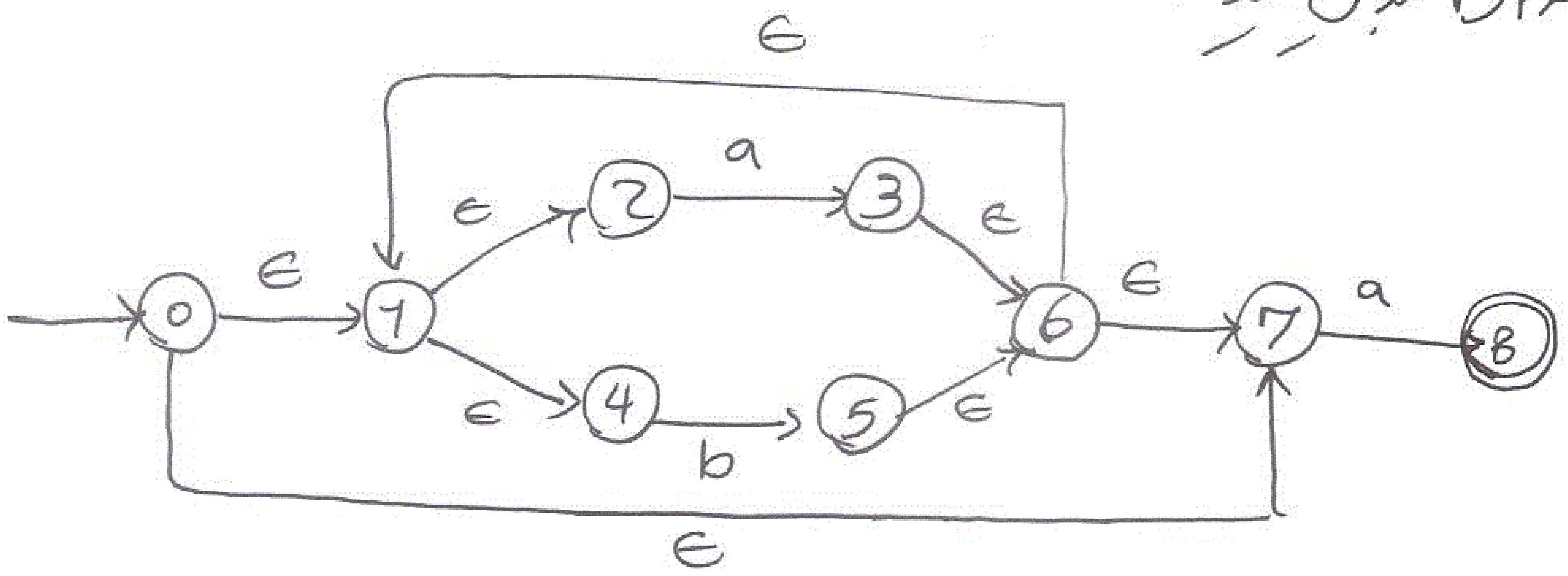
الگوریتم تبدیل NFA به DFA ←

```
initially, E_closure(s0)
while unmarked stat T in Dtrans DO
    begin  mark T; foreach input symbol a do
           begin  U: E_closure(Move(T,a));
                  if (U not Dtrans) add Dtrans;
                  DTrans[T,a]=U
end;
```

جزوه کامپایلر فصل ۲ ص ۸

* تبدیل NFA به DFA

* مثال ۲-۱۸) NFA زیر را به DFA تبدیل کنید:



$(a|b)^*a$ NFA مربوط به

مرحله اول) ε_closure(s) ←

$\epsilon\_closure(0) = \{0,1,2,4,7\} = A$ →

DTrans

حالات	a	b
A		

	a	b
✓A		

مرحله دوم) یک حالت دستنخورده در DTrans می یابیم که در این جا همان A هست (مرحله ۳) ابتدا A را علامت زده و مراحل زیر را دنبال می کنند

مرحله چهارم) برای هر یک از نمادهای a و b مراحل زیر را دنبال می کنند:

۴-۱) $U = \epsilon\_closure(move(A,a))$ را به دست می آوریم.

$move(A,a) = move(\{0,1,2,4,7\},a)$

$= move(0,a) \cup move(1,a) \cup move(2,a)$

$\cup\, move(4,a) \cup move(7,a) = \{\} \cup \{\} \cup \{3\} \cup \{\} \cup \{8\} = \{3,8\}$

$U = \epsilon\_closure(move(A,a)) = \epsilon\_closure(3) = \{3,6,7,1,2,4,8\} = B$

	a	b
✓A	B	C
B		
C		

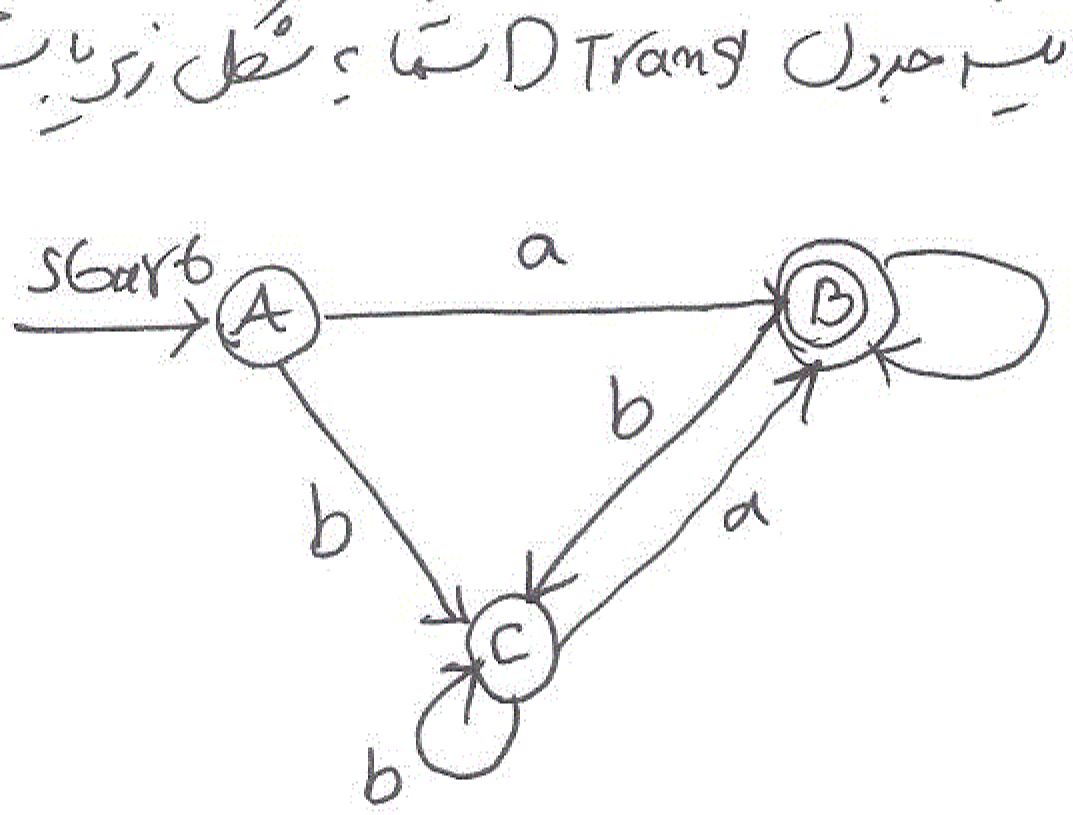
* با توجه به اینکه مجموعه جدید با A مساوی نیست آنرا B نامیده و به جدول DTrans اضافه می کنند در این حالت B علامت نخورده است

۴-۱) $\epsilon\_closure(move(A,b))$ را محاسبه می کنند که برابر با ← $U = \epsilon\_closure(5) = \{5,6,7,1,2,4\} = C$

* با توجه به اینکه مجموعه جدید با A و B مساوی نیست آنرا C نامیده به جدول اضافه می کنند اکنون B و C علامت نخورده اند.

* همین روند را برای کلیه حالتهای علامت نخورده نیز اعمال می کنند

* بقیه حل این مثال طبق حالت A به عهده دانشجو واگذار می شود، در نهایت جدول DTrans شما به شکل زیر باشد.



حالتها	a	b
✓A	B	C
✓B	B	C
✓C	B	C

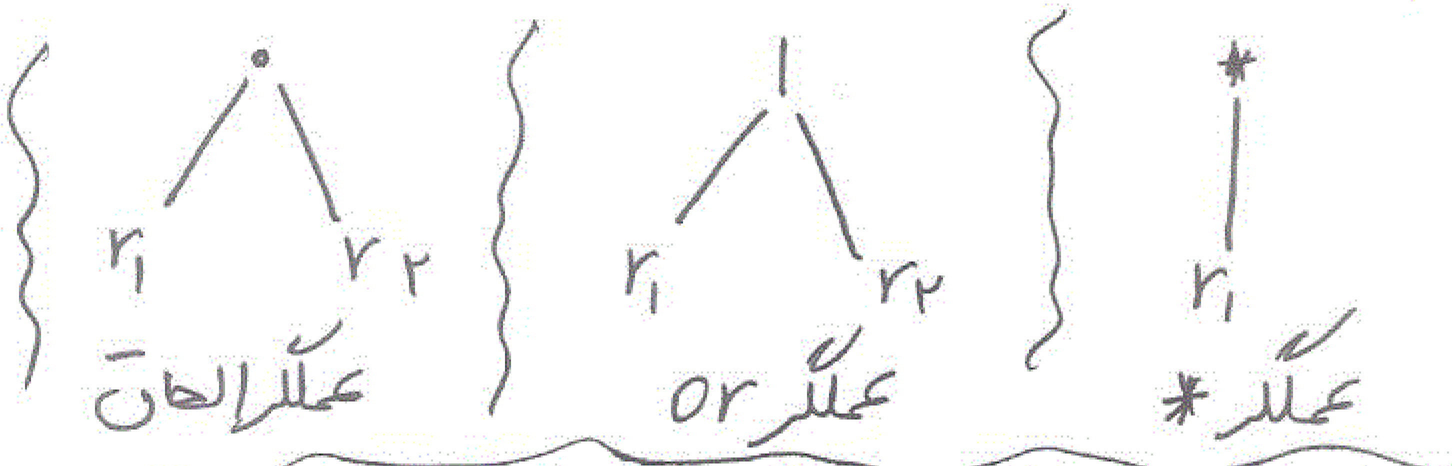
ولذا DFA را رسم می کنند

* دیدید چه باحال بود!؟

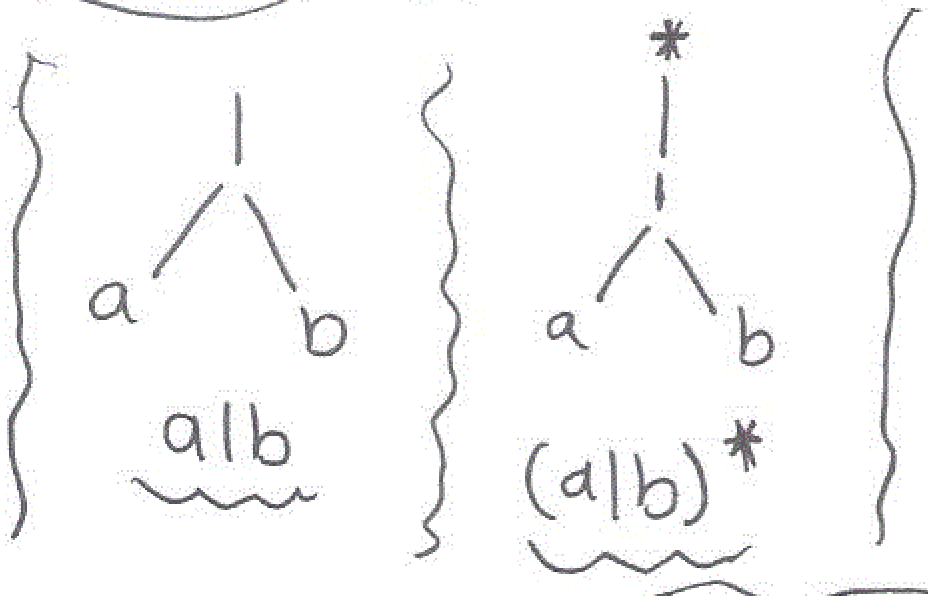
جزوه کامپایلر فصل ۲ ص ۹

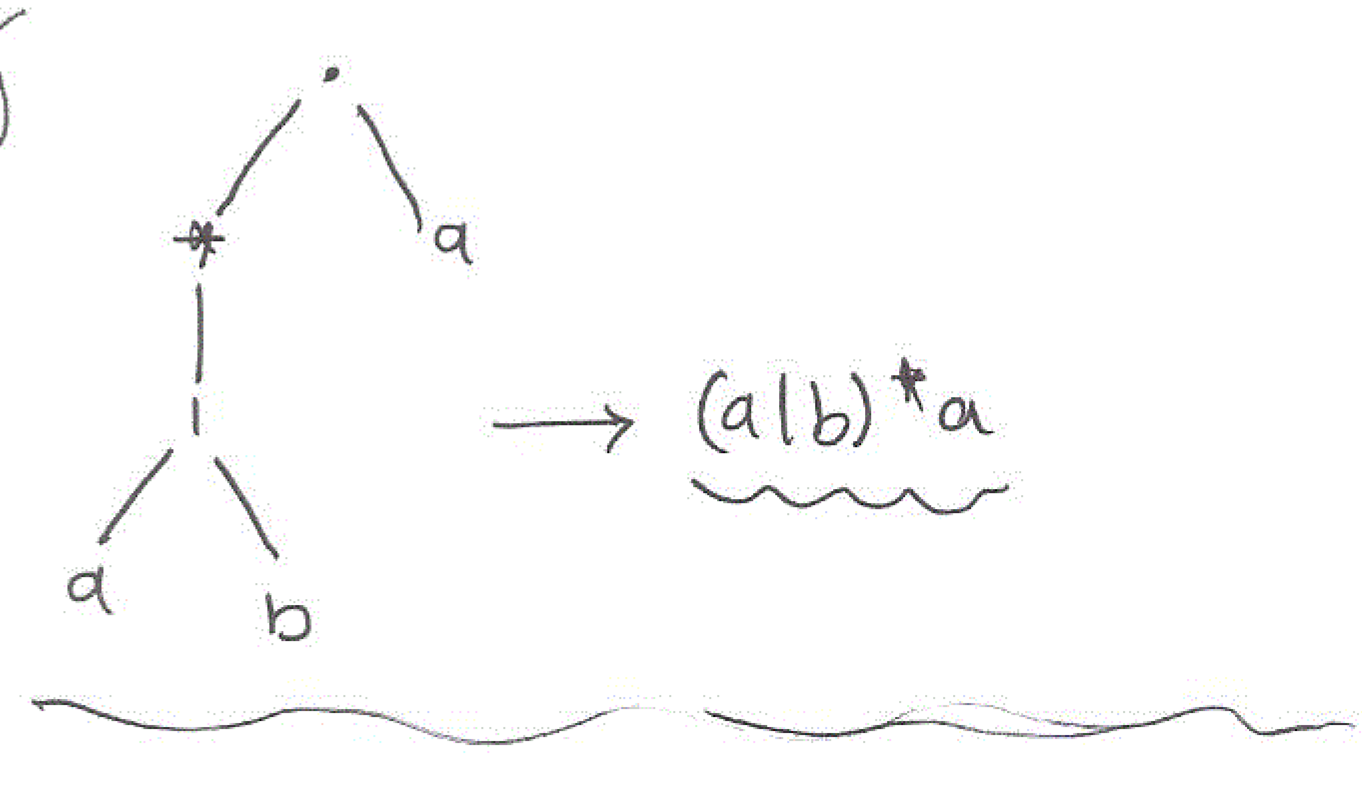
* ساختن مستقیم DFA از عبارت با قاعده

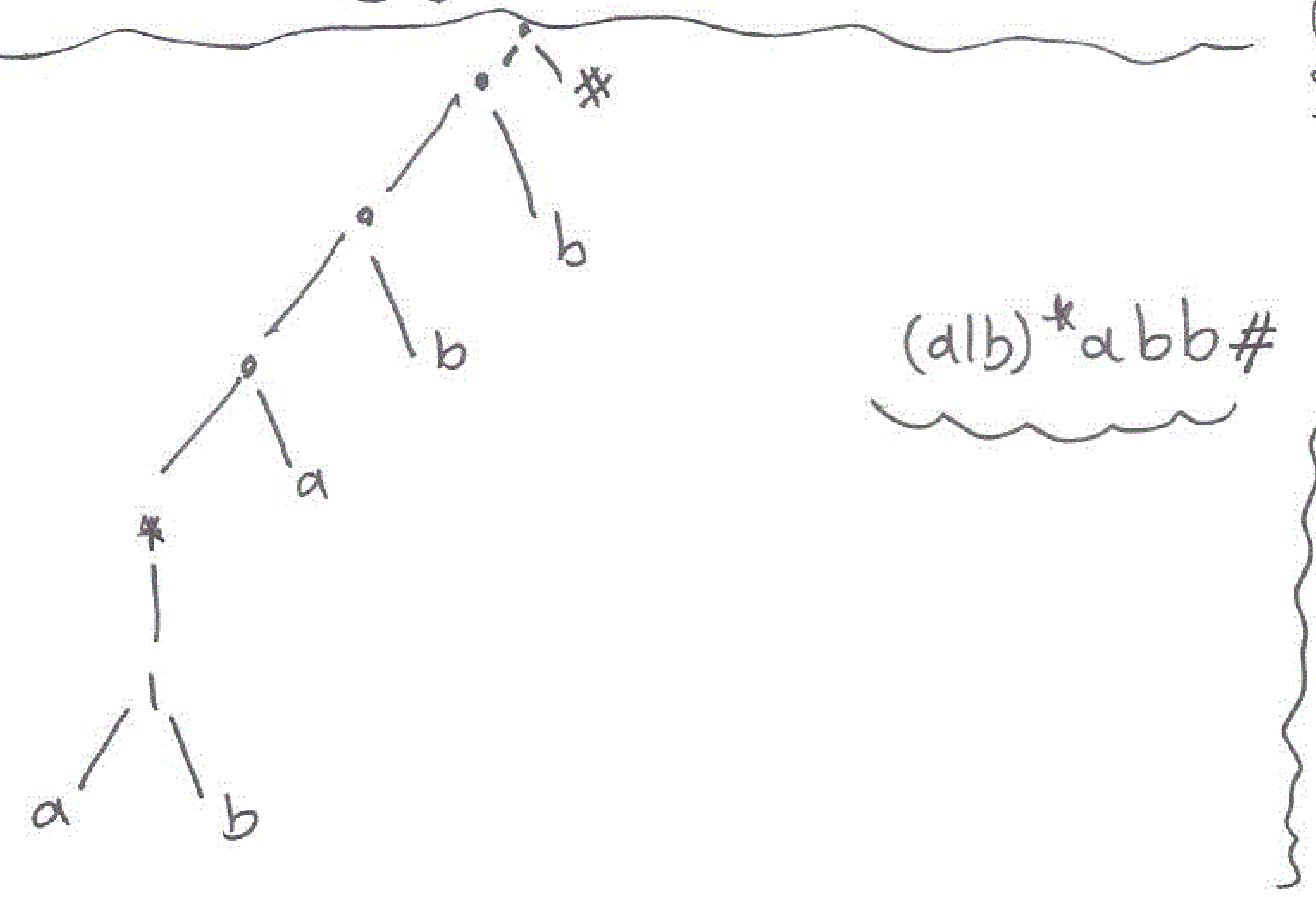
* ابتدا درخت‌های زیر را به منظور درک این قسمت به خاطر بسپارید!

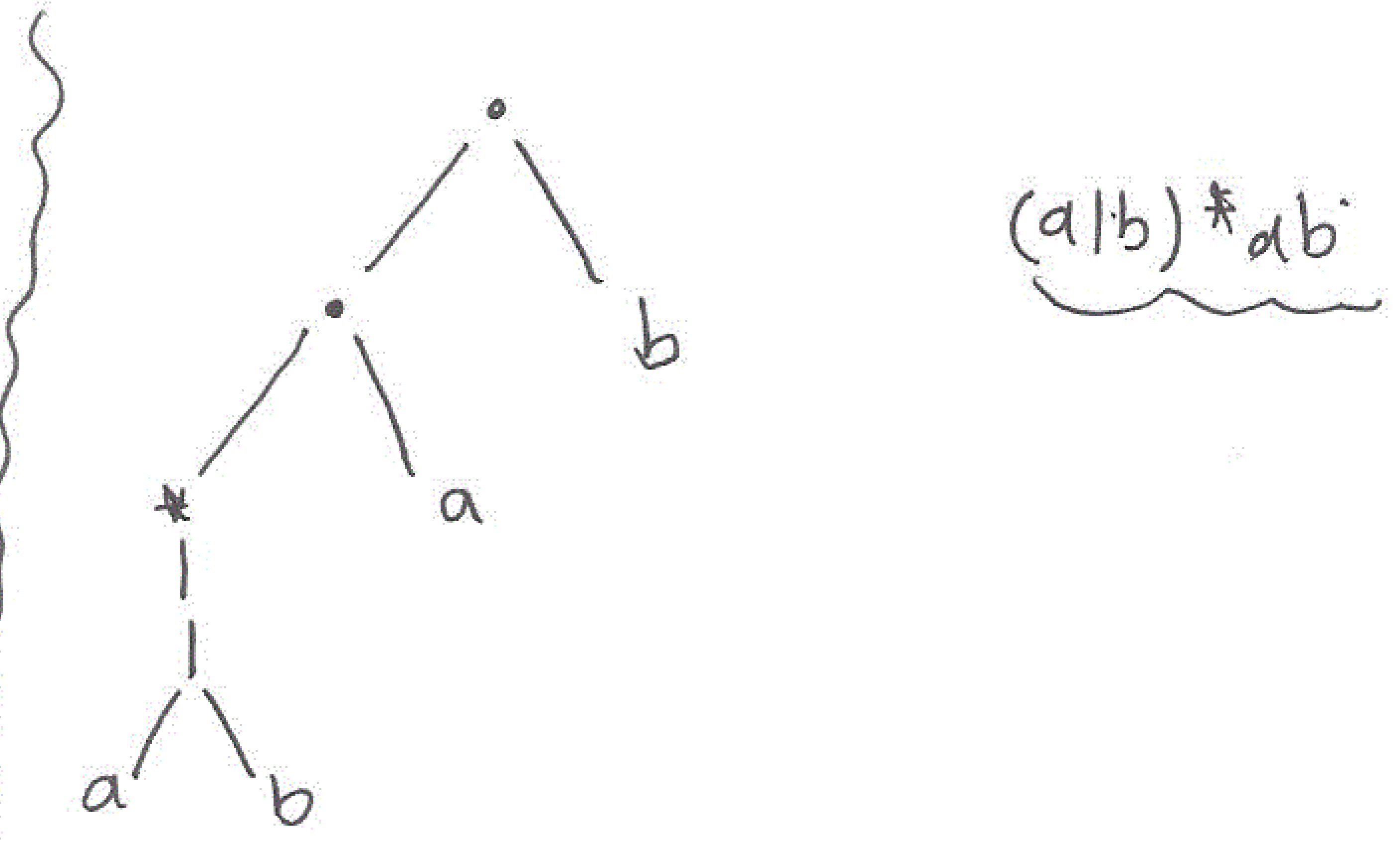


* نماد ε به برگ تبدیل میشود









* حال نیاز به تعاریف زیر داریم: ۱) nullable(n): این تابع مشخص می‌کند آیا زیردرخت n می‌تواند ε را تولید کند یا خیر
اگر بتواند تولید کند مقدار این تابع True و اگر نه false میشود

درمورد nullable قوانین زیر برقرارند: ۱-۱) اگر n برگی با برچسب ε باشد آنگاه nullable(n)، True است

۱-۲) اگر r=a باشد، nullable(n) false است.

۱-۳) اگر r یک ستاره داشته باشد مقدار این تابع True است زیرا عملگر * می‌تواند ε تولید کند

۲) firstpos(n): این تابع مجموعه مکان نمادهایی را برمی‌گرداند که منطبق بر اولین نماد در رشته‌های تولیدی از عبارات با قاعده هستند

* به طور مثال در عبارت (a|b)*c چون هر یک از عبارات a، b و c می‌توانند به عنوان نماد شروع باشند لذا firstpos = {1,2,3}

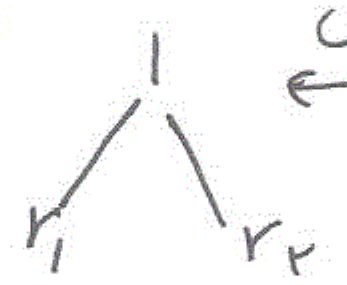
۳) lastpos(n): این تابع مجموعه مکان نمادهایی را برمی‌گرداند که منطبق بر آخرین نماد رشته‌های تولیدی از عبارات با قاعده هستند.

* به طور مثال در عبارت (a|b)*c یا (a|b)+c چون آخرین نماد c است لذا lastpos = {3} می‌شود.

* ساخت مستقیم DFA از عبارات با قاعده

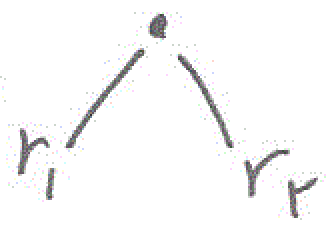
* بین توابع nullable، firstpos و lastpos که در ص قبل یاد شد روابط زیر برقرار است:

Cat = ∘
or = |
star = *



۱) اگر r یک or باشد که سمت چپش r_1 و سمت راستش r_2 باشد آنگاه: ← شکل

* nullable(r) = nullable(r_1) or nullable(r_2)
* firstpos(r) = firstpos(r_1) ∪ firstpos(r_2)
* lastpos(r) = lastpos(r_1) ∪ lastpos(r_2)



۲) اگر r یک Cat باشد که سمت چپش r_1 و سمت راستش r_2 باشد آنگاه:

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

```
if(nullable(r2) = true)
    lastpos(r) = lastpos(r1) U lastpos(r2)
else
    lastpos(r) = lastpos(r2)
```

۳) اگر r یک star باشد آنگاه ← nullable(r) = true, firstpos(r) = firstpos(r_1), lastpos(r) = lp(r_1)

* بعد از بررسی این ۳ تابع و روابط موجود بین آنها، به بررسی تابع چهارم یعنی followpos(i) می‌پردازیم:

۴) followpos: این تابع مجموعه مکان‌هایی را که می‌توانند بعد از مکان i قرار گیرند.

* الگوریتم تبدیل مستقیم DFA از عبارات با قاعده: ۱) درخت با قاعده r# را رسم می‌کنیم

۲) توابع nullable، firstpos، lastpos و followpos را در پیمایش عمقی محاسبه می‌کنیم.

۳) firstpos(root) را محاسبه می‌کنیم (ریشه درخت) و آن را به عنوان یک حالت علامت نخورده به DTrans اضافه می‌کنیم

۴) اگر حالت علامت نخورده‌ای در DTrans موجود است آن را انتخاب می‌کنیم اگر وجود ندارد الگوریتم پایان یافته است.

۵) حالت انتخاب شده را T نامیده و آن را تیک می‌زنیم ۶) برای هر نماد a مراحل ذیل را اجرا می‌کنیم

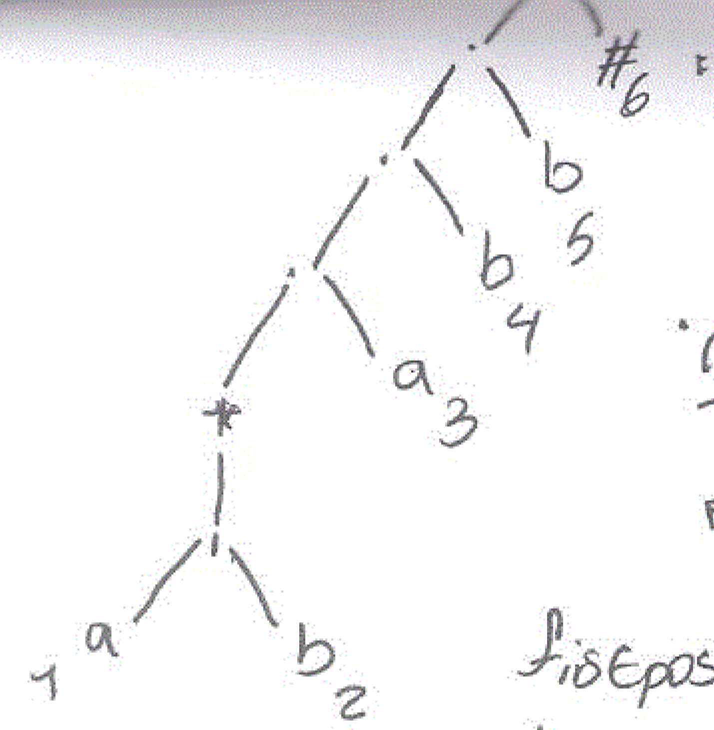
۶-۱) مجموعه مکان‌هایی در T که نماد متناظر در عبارت با قاعده a هستند را محاسبه کرده و آن را p می‌نامیم

۶-۲) مجموعه followpos(p) را محاسبه نموده و آن را U می‌نامیم ۶-۳) U را به جدول اضافه می‌کنیم ۶-۴) dT[T,a] = U

۷) به مرحله ۴ برگرد ۸) حالتی که شامل مکان متناظر با # است حالت پذیرش DFA است.

* برای درک بهتر الگوریتم در ص ۱۱ جزوه یک مثال را حل می‌کنیم:

* مثال ساخته مستقیم DFA از عبارت باقاعده:



مثال) می‌خواهیم عبارت باقاعده $r=(a|b)^*abb$ را به DFA تبدیل کنید به روش مستقیم.

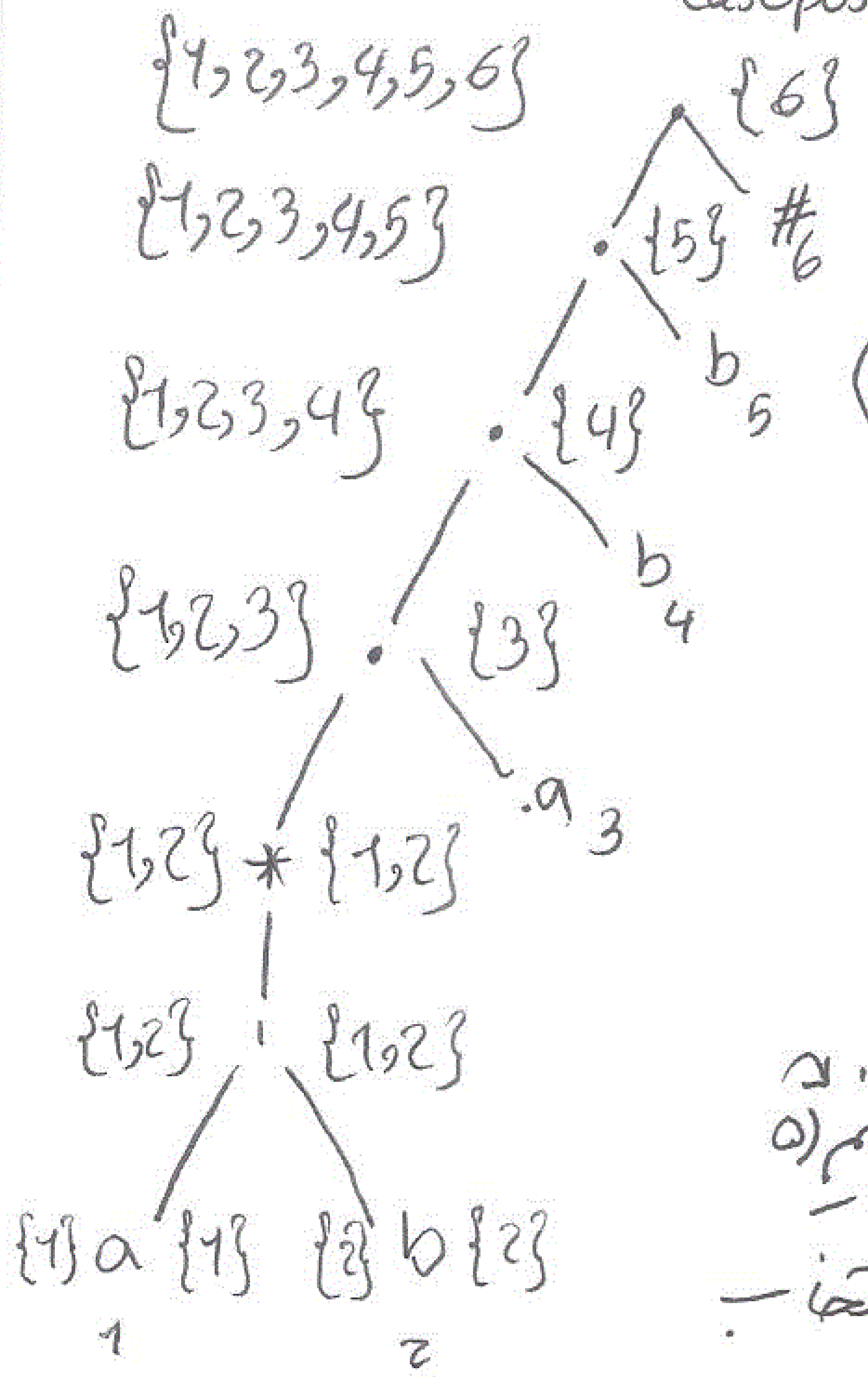
حل. ابتدا علامت # را به آخر عبارت باقاعده اضافه می‌کنیم و سپس درخت آن را رسم می‌کنیم ←

۲) با توجه به مرحله دوم الگوریتم first و last را برای این درخت محاسبه می‌کنیم برای (a)

firstpos(1)={1}
lastpos(1)={1}

* برای تمامی نمادها نیز به همین شکل . به صورت یک مجموعه (first در چپ و last در راست)

درکنار درخت می‌نویسیم. در نتیجه درخت ما به شکل روبرو تبدیل می‌شود ←



* سپس با استفاده از firstpos و lastpos بدست آمده followpos را بدست می‌آوریم

مکان	Followpos
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	—

۳) firstpos ریشه را بدست می‌آوریم A={1,2,3}

A را به جدول Dtrans اضافه می‌کنیم

حالت	a	b
A		

۵) یک حالت علامت نخورده (A) را از Dtrans انتخاب می‌کنیم و آن را علامت می‌زنیم

۶) برای نماد a مراحل زیر را دنبال می‌کنیم: ۶-۱) برای نماد a از مجموعه A مکان‌هایی را انتخاب

می‌کنیم که نماد متناظر آنها در عبارت باقاعده a است در این صورت p={1,3} میشود

* اگر در شکل درخت دقت کنید تنها برای اعداد ۱ و ۳ عبارت a را شاهد هستید*

حالت	a	b
✓A	B	A
B		

۶-۲) {1,2,3,4}=followpos(1) U followpos(3) چون این مجموعه با مجموعه‌ای قبلی متفاوت است آن را به جدول اضافه می‌کنیم

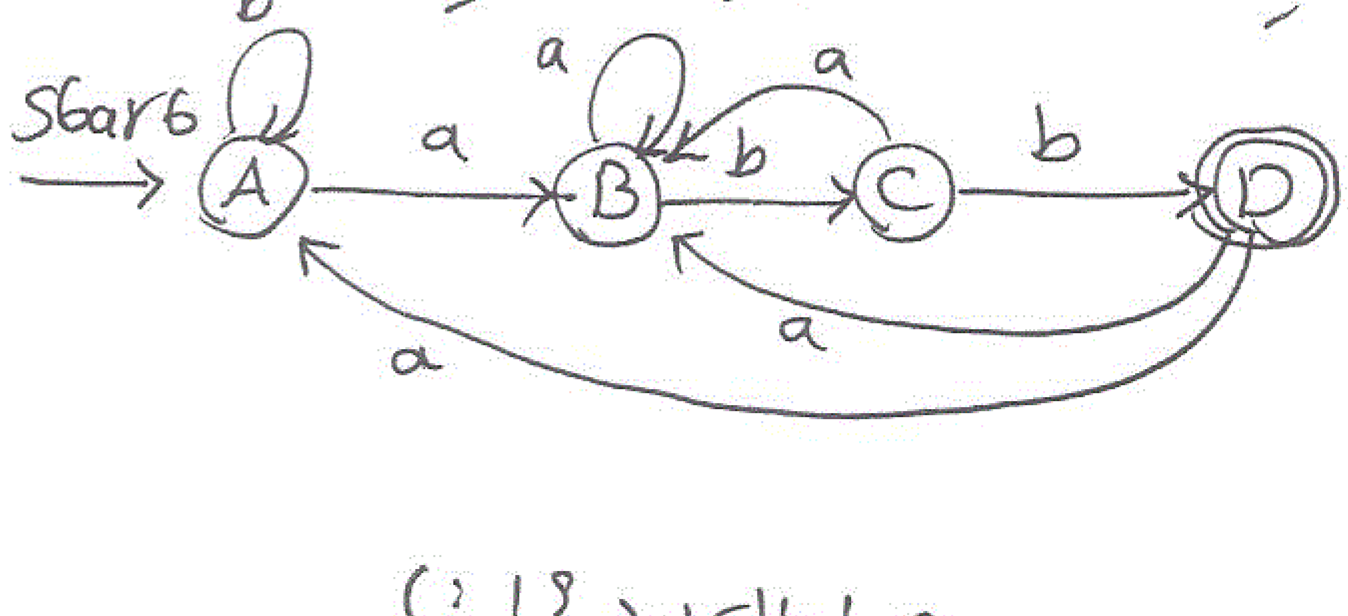
فقط آن را اضافه می‌کنیم ←→ dtrams[A,B]=A

۶-۳) B را به جدول اضافه می‌کنیم ۶-۴) DTrans[A,a]=B قرار می‌دهیم

* با توجه به مرحله ۶ برای نماد B نیز مراحل بالا را دنبال می‌کنیم ← p={2} ← follow(2)=A آن را اضافه نمی‌کنیم به جدول حالت

* با توجه به مرحله چهارم الگوریتم یک حالت علامت نخورده (B) را انتخاب نموده و مراحل بالا را بر روی آن اعمال می‌کنیم

* تقسیم حل این مثال تا حالت A به عهده دانشجو واگذار شده میشود در نهایت DTrans شما باید به شکل زیر باشد:



DFA تولیدی ←

حالت	a	b
✓A	B	A
✓B	B	C
✓C	B	D
✓D	B	A

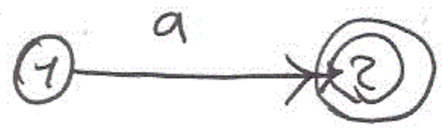
* دیدید چه باحال بود؟! (:

* پیاده سازی کامپایلر: الگوریتم پیاده سازی

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

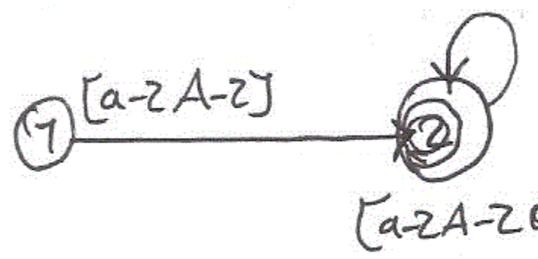
مثال: $r=a$

حل: ابتدا DFA مربوط به این عبارت را رسم می کنیم:



و سپس الگوریتم آن را می نویسیم ←

* مثال: $r=[a-zA-Z][a-zA-Z0-9]^*$ ← DFA

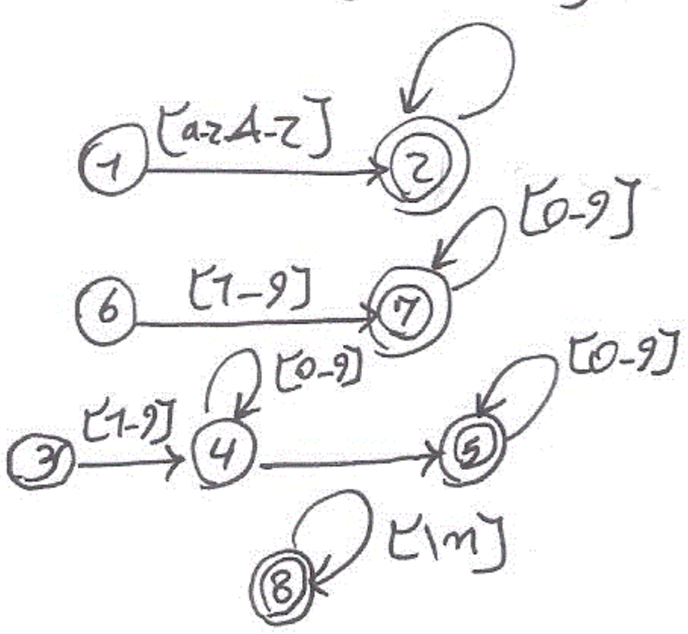


در این حالت نیز الگوریتم دارای ۲ حالت خواهد بود فقط شرط if اول به صورت زیر تغییر می کند و بقیه الگوریتم دست نخورده باقی می ماند

```
if(ch>'a'&&'z'>=ch)||(ch>'A'&&'Z'>=ch)
```

در انتهای الگوریتم نیز هنگام شرط ها به همان تک OR دیگر قرار داده و در صورت برقرار بودن دارد $state=2 شده در غیر اینصورت خارج میشود ← ||(ch>='0'&&'9'>=ch)

* مثال:



۱) شناسه ← عبارت باقاعده ← $r_1=[a-zA-Z][a-zA-Z0-9]^*$ ← DFA

۲) عدد صحیح ← 〃 ← $r_2=[1-9][0-9]^*$ ← 〃

۳) عدد اعشاری ← 〃 ← $r_3=[1-9][0-9]^*.[0-9]^*$ ← 〃

۴) فضای خالی ← 〃 ← $r_4=[\n]^*$ ← 〃

* کل این حالت ها را می توان در یک الگوریتم با چهار حالت Case نوشت.

* کلمات کلیدی (تشخیص)

* برای تشخیص کلمات کلیدی دو روش وجود دارد: ۱) با هر کلمه کلیدی، به شکل مستقل برخورد می کنیم، یک عبارت با قاعده در نظر گرفته و سپس DFA آن را بدست می آوریم و آن را با دیگر DFA ها پیاده سازی می کنیم. در این روش باید ابتدا نمادهای ورودی با عبارات با قاعده مربوط به کلمات کلیدی و سپس با عبارات با قاعده مربوط به دیگر انواع لغات مقایسه شود.

۲) از آنجا که کلمات کلیدی از لحاظ ساختار زیرمجموعه شناسه ها هستند می توان آنها را مانند شناسه ها تشخیص داد با این تفاوت که کلمات کلیدی در جدول نمادها به عنوان مقدار اولیه ثبت می شوند و نشانه کلمات کلیدی نیز در جدول نمادها ثبت می شود

* تولید خودکار تحلیلگر لغوی - روشها: ۱) استفاده از زبانهای برنامه سازی: این روش دارای مزایا و معایبی هست که عبارتند از:

الف) صرف زمان زیاد ب) کاهش قابلیت استفاده مجدد ج) افزایش امکانات د) افزایش قابلیت انعطاف پذیری

۲) استفاده از ابزارها: این روش نیز دارای مزایا و معایبی هست که عبارتند از:

الف) افزایش سرعت ایجاد تغییرات ب) کاهش زمان ساخت ج) افزایش محدودیتها

* flex برنامه ای است که تحلیلگر لغوی تولید می کند. flex یک کامپایلر است که برنامه به زبان flex را به زبان C یا پاسکال تولید می کند. پسوند این فایل (l.) است و با کامپایل برنامه C تحلیلگر لغوی تولید می گردد

«پایان فصل دوم»

فصل سوم: «تحلیلگر نحوی»

* تحلیلگر نحوی علاوه بر بررسی صحت ترتیب لغات برنامه مبدأ، وظیفه ایجاد درخت تجزیه را نیز به عهده دارد. ورودی تحلیلگر نحوی دنباله ای از نشانه هاست که توسط تحلیلگر لغوی ایجاد شده است و خروجی تحلیلگر نحوی درخت تجزیه است که نشان دهنده ساختار برنامه مبدأ می باشد. البته این درخت به شرطی ایجاد میشود که ساختار نحوی برنامه مبدأ صحیح باشد.

* ساخت تحلیلگر نحوی نیازمند توصیف دقیق و کامل ساختار نحوی برنامه مبدأ است و این ساختار به وسیله قوانین بازگشتی تعریف میشود

* یکی از روشهای مهم بیان قواعد نحوی زبان مبدأ، استفاده از گرامرهای مستقل از متن است. این گرامرها مزایای زیر را در تحلیل نحوی دارند:

۱) گرامرهای مستقل از متن، قوانین نحوی زبان مبدأ را کامل، دقیق و قابل فهم توصیف می کنند

۲) با استفاده از گرامرهای مستقل از متن می توان ابزارهایی را ایجاد کرد که گرامر مستقل از متن را گرفته و تحلیلگر نحوی آن را به صورت خودکار تولید کنند. که البته تولید خودکار همه گرامرهای مستقل از متن را شامل نمی شود

جزوه کامپایلر   فصل سوم   ص ۱۴

* مدیریت خطا : تحلیل گر لغوی ← تحلیل گر نحوی

* خطاهای برنامه در سطوح مختلفی رخ می دهند که عبارتند از: ۱) خطاهای لغوی ۲) خطاهای نحوی مثل پرانتزهای نامتعادل، سمت چپ غیرمجاز، استفاده نادرست از عملگرها ... ۳) خطاهای معنایی: این نوع خطاها توسط عبارات با قاعده و گرامرهای مستقل از متن قابل بیان نیستند این نوع خطاها را تحلیل گر معنایی کشف می کند مثل: نوع داده، حوزه کنترل، کنترل یکتایی

۴) خطاهای منطقی: مربوط به منطق برنامه است و کشف آن برعهده برنامه نویس است مثل اشکال در الگوریتم، ایجاد حلقه های بی نهایت

* پیغام خطا باید شامل دو قسمت باشد که عبارتند از: ۱) محل خطا، ۲) راهنمایی جهت رفع خطا.

* در مورد خطاهای لغوی تعیین محل خطا امکان پذیر است اما تعیین محل دقیق خطاهای نحوی به سادگی امکان پذیر نیست.

* گرامر مستقل از متن: ابزار مناسبی جهت توصیف ساختار نحوی زبانهای برنامه سازی است و متشکل از مجموعه ای از قواعد تولید و یک نماد شروع است. هر قاعده تولید، دارای ۲ قسمت است. بخش سمت چپ که نام ساختار است و بخش سمت راست است که تمام شکلهای ممکن ساختار را نمایش می دهد. بخش سمت راست می تواند شامل نمادهای پایانه و غیر پایانه باشد، پایانه ها و غیر پایانه ها را نمادهای گرامر نامیده، در نوشتن گرامرهای مستقل از متن اصول زیر رعایت می گردد تا درک ساده تری داشته باشند:

۱) پایانه ها را با یکی از صورتهای ← - حروف کوچک - عملگر و علائم - ارقام - نشان می دهند.

۲) غیر پایانه ها را با یکی از صورتهای ← - حروف بزرگ - اسامی با حروف کوچک مثل expr, start

۳) برای نمایش دنباله ای از پایانه ها و غیر پایانه ها از حروف لاتین مثل α و β استفاده می کنیم.

۴) سمت چپ اولین قاعده تولید به عنوان نماد شروع در نظر گرفته میشود

* برای تشریح ساختار نحوی زبان برنامه نویسی از نوع خاصی از گرامر مستقل از متن به نام BNF استفاده می کنند که ویژگیهای زیر را دارد:

۱) سمت چپ و راست قانون با ::= از یکدیگر جدا میشوند ۲) غیر پایانه ها درون < > قرار می گیرند ۳) انتخابها با | از هم جدا میشوند

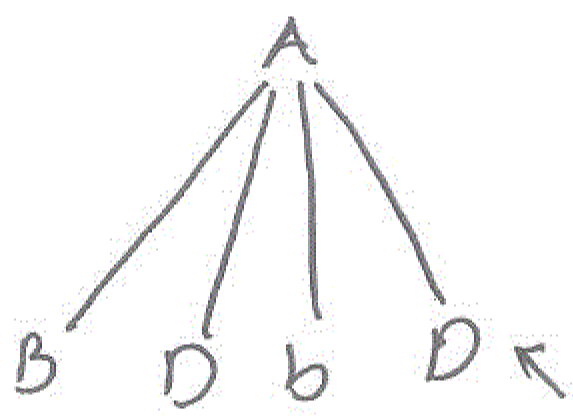
* امروزه نوع دیگری از BNF به نام EBNF وجود دارد که تفاوتهایی به شرح ذیل با BNF دارد:

۱) پایانه ها درون علامت " " قرار می گیرند، ۲) نمادهای اختیاری داخل علامت [ ] قرار می گیرند

۳) برای نمایش تکرار صفر یا بیشتر از * ۴) برای نمایش تعداد یک یا بیشتر از + استفاده میشود (مثل عبارات با قاعده)

۵) قسمتهای تکراری را فاکتورگیری کرده و در علامت { } قرار می گیرد.

* درخت تجزیه * ساختار نحوی برنامه مبدا را نمایش می دهد *



* درخت تجزیه چگونگی تولید یک رشته از نماد شروع گرامر را به صورت تصویری نشان میدهد مثال $A \to BDbD$

* روابط برقرار بین درخت تجزیه و گرامر مستقل از متن عبارتند از: ۱) ریشه درخت تجزیه متناظر با نماد شروع گرامر است.

۲) برگهای درخت تجزیه، پایانه گرامر یا ϵ هستند ۳) گره داخلی درخت متناظر با یک غیرپایانه است.

۴) اگر $A \to \epsilon$ یعنی گره A می تواند فرزند ϵ داشته باشد.

* اشتقاق: فرایند تولید یک رشته از نماد شروع را اشتقاق گوییم و در هر مرحله یک غیرپایانه بوسیله سمت راست قاعده تولید گسترش داده میشود و این روند آنقدر ادامه می یابد تا هیچ غیرپایانه ای باقی نماند. هر دنباله که در هر مرحله از اشتقاق تولید میشود را یک شبه جمله گوییم.

* انتخاب شبه جمله و قواعد تولید انتخابی آزاد است ولی اگر در هر مرحله سمت چپ ترین غیرپایانه گسترش داده شود اشتقاق را سمت چپ ترین اشتقاق و اگر سمت راست باشد اشتقاق را سمت راست ترین گویند.

* تفاوت درخت تجزیه و اشتقاق در این است که درخت تجزیه ترتیب گسترش غیرپایانه ها را نمایش نمی دهد اما در اشتقاق مهم است.

* گرامرهای مبهم: گرامری که بتوان برای رشته تولید شده از گرامر بیش از یک درخت تجزیه تولید کرد. روش های رفع ابهام:

۱) تغییر گرامر ۲) استفاده از قوانین جانبی مثل استفاده از اولویت - استفاده از قوانین خاص

* بازگشتی چپ: اگر سمت راست قاعده تولید با غیرپایانه سمت چپ قاعده تولید شروع شود در این صورت گرامر دارای بازگشتی چپ است

مثل $A \to A+T$ یا $X \to XT$ * بازگشتی چپ به ۳ دسته تقسیم میشود ۱) بازگشتی چپ مستقیم ۲) غیرمستقیم ۳) مخفی (ϵ)

* فاکتورگیری چپ: برای یک غیرپایانه ممکن است انتخاب های مختلفی وجود داشته باشد بطوریکه شروع یکسان داشته باشند، چنین گرامرهایی درخت تجزیه کننده را مشکل می کنند. برای حذف این خاصیت از فاکتورگیری چپ استفاده می کنند، به این شکل که:

برای غیرپایانه ای مانند A که دارای چندین انتخاب است، طولانی ترین رشته مشترک ابتدای انتخاب های مختلف سمت راست قواعد تولید A را می یابیم و این قسمت را با α نمایش می دهیم و قسمت غیرمشترک را با β نمایش می دهیم ⇐ $A \to \alpha\beta_1 | \alpha\beta_2 \dots \alpha\beta_n$

گرامر فوق را می توان به شکل $A \to \alpha R$ و $R = \beta_1 | \beta_2 \dots \beta_n$ نمایش داد

مثال: گرامر ذیل را در نظر میگیریم

$$A \to cdB \mid cdgD$$
$$B \to bB \mid c$$
$$D \to dD \mid e$$

با توجه به $A \to cdB \mid cdgD$ قسمت مشترک cd است

لذا ⇐ $\alpha = cd$ ، $\beta_1 = B$ ، $\beta_2 = gD$ ⇐

$$A \to cdR$$
$$R \to B \mid gD$$
$$B \to bB \mid c$$
$$D \to dD \mid e$$

گرامر نهایی با فاکتورگیری چپ

* رابطه تحلیل‌گر لغوی و تحلیل‌گر نحوی *

* تحلیل‌گر لغوی به وسیله یک تابع پیاده‌سازی می‌گردد که توسط تحلیل‌گر نحوی فراخوانی می‌شود. تحلیل‌گر لغوی در هر فراخوانی برنامه را بررسی کرده و یک نشانه استخراج و به تحلیل‌گر نحوی ارسال می‌کند. نشانه‌های بازگشتی از تحلیل‌گر نحوی با پایانه‌های گرامر مستقل از متن مورد استفاده تحلیل‌گر نحوی است. تحلیل‌گر نحوی با توجه به گرامر، هر جایی که نیاز به یک پایانه باشد تحلیل‌گر لغوی را فراخوانی می‌کند و درخواست نشانه بعدی را می‌کند اگر نشانه دریافتی همان نشانه مورد انتظار باشد که کار خود را ادامه می‌دهد وگرنه خطا رخ داده است.

* تجزیه: تحلیل‌گر لغوی برنامه مبدأ را به دنباله‌ای از نشانه‌ها تبدیل کرده و به تحلیل‌گر نحوی ارسال می‌کند. این فرایند را تجزیه گوییم. در اصل تجزیه مشخص می‌کند نشانه دریافتی از تحلیل‌گر لغوی توسط گرامر زبان قابل تولید هست یا خیر.

* انواع تجزیه کننده‌ها: ۱) بالا به پایین ۲) پایین به بالا   * همه تجزیه‌کننده‌ها با استفاده از گرامر برای رشته ورودی درخت تجزیه می‌سازند.

* تجزیه کننده‌های بالا به پایین گره‌های درخت تجزیه را به ترتیبی که در پیمایش preorder و پایین به بالا در پیمایش Posorder ایجاد می‌کنند.

* در تجزیه کننده‌های بالا به پایین، یک گره ایجاد و سپس گره‌های فرزندان از چپ به راست ایجاد می‌گردند و در تجزیه کننده‌های پایین به بالا ابتدا فرزندان گره و سپس خود گره ایجاد می‌شود. در هر دو تجزیه‌کننده رشته ورودی را از چپ به راست پویش می‌کنند

* اگر مراحل ایجاد درخت تجزیه توسط تجزیه‌کننده پایین به بالا را از آخر به اول مرور کنیم مشاهده می‌شود که سمت راست‌ترین اشتقاق تولید شده است، به عبارت دیگر هر مرحله از تجزیه پایین به بالا عکس عمل سمت راست‌ترین اشتقاق است، بنابراین اولین گره‌ای که در تجزیه پایین به بالا تولید می‌شود مطابق آخرین نمادی است که در سمت راست‌ترین اشتقاق گسترش می‌یابد

* تجزیه کننده بالا به پایین در هر مرحله یک غیرپایانه را با سمت راست قاعده تولید آن گسترش می‌دهد تا در نهایت به رشته مورد نظر برسد در حالیکه تجزیه کننده پایین به بالا در هر مرحله سمت راست یک قاعده تولید را در دنباله ورودی آن یافته، آن را به سمت چپ قاعده تولید کاهش می‌دهد تا در نهایت نماد شروع ایجاد شود.

* تجزیه کننده‌های بالا به پایین را می‌توان به روش دستی ایجاد کرد، حال آنکه تجزیه کننده‌های پایین به بالا به قدری پیچیده است که ایجاد آن فقط با استفاده از ابزارهای تولید تجزیه کننده منطقی است.

* به منظور تشریح این تجزیه کننده‌ها، ۲ مفهوم first و follow را در صفحه بعد تشریح می‌کنیم که بسیار ضروری و مهم هستند.

* از این جا به بعد مباحث کمی نیاز به دقت بیشتری دارد....

* تجزیه کننده

① * first (α): اگر α دنباله‌ای از پایانه و غیرپایانه‌ها باشد، مجموعه first مربوط به α پایانه‌هایی را مشخص می‌کند که رشته‌های مشتق شده از α با آنها شروع می‌شود. اگر α بتواند ∈ را تولید کند، ∈ به first(α) اضافه می‌شود.

مثال. با توجه به گرامر رو‌برو first (α) را بیابید.

$A \to Bcd$
$B \to bB \mid e \mid \epsilon$
$C \to aC \mid \epsilon$

گرامر

حل. ابتدا همه رشته‌هایی که از Bcd مشتق شده‌اند را حساب می‌کنیم

① اگر B و C را ∈ فرض کنیم ← $BCd \to d$

یا ② $Bcd \to bBCd$ ← B را bB فرض کرده‌ایم
bBd ← C را ∈ فرض کرده‌ایم
bd ← B را ∈ فرض کرده‌ایم

یا ③ $BCD \to eCd$ B را e فرض کرده‌ایم
ed C را ∈ فرض کرده‌ایم

یا ←

④ $Bcd \to cd$ ← B را ∈ فرض کرده‌ایم
acd ← C را aC فرض کرده‌ایم
ad ← C را ∈ فرض کرده‌ایم

با توجه به مشتق‌ها متوجه می‌شویم که شروع تمام رشته‌های مشتق شده {b و e و a} و d هستند یعنی first(α) = {a, e, b, d} هستند

② * follow(A): پایانه‌هایی را مشخص می‌کند که در اشتقاق‌های مختلف بلافاصله در سمت راست A قرار می‌گیرند.

مثال. با توجه به گرامر ذیل follow(A) را بدست آورید. حل ←

گرامر:

$$\begin{cases} X \to aXAad \mid a \\ A \to Ac \mid Af \mid \epsilon \end{cases}$$

$X \to aXAad$
$X \to aXAad \to aXAcad$
$X \to aXAad \to aXAcad \to aXAfcad$

یعنی پاسخ {f و c و a} می‌باشد

* جهت سهولت یافتن follow می‌توان از قوانین ذیل استفاده نمود:

۱) اگر S نماد شروع باشد ($ نشان دهنده آخر رشته ورودی است) به follow(S) اضافه می‌شود.

۲) برای هر قاعده تولیدی به صورت $M \to \alpha N \beta$ تمام پایانه‌های موجود در first(β) به جز ∈ به follow(N) اضافه می‌شود

۳) برای هر قاعده تولیدی به صورت $M \to \alpha N$ و یا $M \to \alpha N \beta$ در صورتیکه ∈ عضو first(β) باشد یعنی β بتواند رشته تهی یا ∈ را تولید کند تمام پایانه‌های مجموعه follow(M) به follow(N) اضافه می‌شود

جزوه کامپایلر فصل ۳ ص ۱۸ * تجزیه کننده‌ها + مثال یافتن follow‌ها با استفاده از قوانین

مثال: با توجه به گرامر زیر follow(B) را بیابید.

گرامر:

$A \to AXb$
$X \to d \mid dB \mid eBE$
$E \to a \mid \epsilon$
$B \to b$

حل: با استفاده از AXb و با استفاده از قانون ۲ یا مرحله‌ی ۲ first(b) به follow(X) اضافه می‌شود {b}

با توجه به eBE و قانون ۲ یا مرحله‌ی ۲ first(E) به جز ε به follow(B) اضافه می‌گردند {a}

با توجه به X→dB و follow(X)={b} با قانون ۳ یا مرحله‌ی ۳ به X ، follow(B) " followB={a,b}

* تجزیه کننده‌ی بالا به پایین: تجزیه بالا به پایین از نماد شروع گرامر آغاز میشود. از یک متغیر پیشنگر به نام lookahead برای تشریح استفاده میشود که این متغیر حاوی نخستین چپ‌ترین پایانه‌ی رشته ورودی هست و در هر مرحله تجزیه سمت چپ درخت را گسترش می‌دهیم.

* تجزیه کننده‌ی پیشگو: نوع خاصی از تجزیه کننده‌ی بازگشتی کاهشی است که در آن اجرای توابع بازگشتی، ورودی را پردازش می‌کنند.

تجزیه کننده‌ی بازگشتی کاهشی از روش سعی و خطا برای پردازش استفاده می‌کند و بسیار کند و زمان‌بر است.

* تجزیه کننده‌ی پیشگوی غیربازگشتی: در این تجزیه کننده با استفاده از پشته و جدول تجزیه بدون فراخوانی بازگشتی توابع رشته‌ی ورودی پردازش میشود

* تجزیه کننده‌ی پیشگو: در تجزیه کننده‌ی پیشگو، توابع بازگشتی، وظیفه‌ی پردازش رشته‌ی ورودی را بر عهده دارند و برای هر غیرپایانه یک تابع ایجاد میشود.

مثال: فرض کنید می‌خواهیم برای گرامر ذیل تجزیه کننده‌ی پیشگو ایجاد کنید.

گرامر:

$A \to B \mid C$
$B \to bB \mid f$
$C \to eC \mid e$

حل: برای پیاده‌سازی این تجزیه کننده سه تابع A() ، B() ، C() مورد نیاز است.

با توجه به A→B|C ، غیرپایانه A باید بین دو تابع B() و C() یکی را انتخاب کند

* نحوه انتخاب بین دو تابع B() و C() به این شکل است که اگر lookahead در {b,f} first(B) باشد B و اگر lookahead در {e} first(C) باشد تابع C() فراخوانی می‌گردد. الگوریتم فوق بصورت زیر است

```
Void A(){
 if(lookahead =='b'|| lookahead =='f')
   B();
 else if (lookahead=='c'|| lookahead=='e')
   C();
 }
 else{cout<<"error";
      exit(0);
  }
 return;
 }
```

* سپس برای پیاده‌سازی B() نیز بین B→bB و B→f یکی باید انتخاب شود و برای C() نیز بین C→eC و C→e یکی باید انتخاب شود و فقط بنا به [illegible] شرط الگوریتم را تغییر داد.

* برای هر گرامری نمی‌توان تجزیه کننده‌ی پیشگو ساخت در حقیقت برای گرامرهایی که دارای بازگشتی چپ هستند نمی‌توان تجزیه کننده‌ی پیشگو ایجاد کرد، مگر اینکه قبلاً بازگشتی چپ به روشی که گفته شد، حذف شود و به این شکل first/first‌ها نداشته باشند.

* تجزیه کننده پیشگوی غیربازگشتی: در این روش برای هر غیرپایانه متناسب با هر پایانه ورودی، قاعده تولید مناسب را در جدولی ذخیره می‌کنیم و تجزیه کننده از این جدول برای انتخاب قاعده تولید مناسب استفاده می‌کند. گرامر ذیل که برای آن تجزیه کننده پیشگو ارائه گردید را در نظر می‌گیریم ↓

$A \to B | C$
$B \to bB | f$
$C \to cC | e$

حل. در بخش قبل مشخص شد که با توجه به first‌ها می‌توان انتخاب کرد که کدام قواعد انتخاب شوند. اگر آنها را در جدولی قرار دهیم ← به این شکل می‌شود:

	b	f	c	e
A	$A \to B$	$A \to B$	$A \to C$	$A \to C$
B	$A \to bB$	$B \to f$		
C			$C \to cC$	$C \to e$

این جدول در روند تجزیه مورد استفاده قرار می‌گیرد، برای تبدیل برنامه بازگشتی به غیربازگشتی از پشته استفاده می‌شود.

* قسمتهای مختلف تجزیه کننده پیشگوی غیربازگشتی عبارتند از: ۱- رشته ورودی ۲- پشته ۳- جدول تجزیه ۴- برنامه تجزیه ۵- خروجی

این تجزیه کننده دارای دو انتقال به نام انتقال پیشگو و انتقال تطبیق است.

* ۱) انتقال پیشگو: اگر بالای پشته غیرپایانه X و نماد جاری رشته ورودی a باشد، X از پشته حذف و قاعده تولید $M[X, a]$ از جدول استخراج شده و به صورت معکوس در پشته قرار می‌گیرد. به عنوان مثال اگر $M[X, a] = X \to \alpha\beta\gamma$ باشد، α در بالای پشته قرار می‌گیرد

* ۲) انتقال تطبیق: اگر نماد بالای پشته، پایانه باشد و این پایانه با نشانه جاری رشته ورودی یکسان باشد، آنگاه پایانه بالای پشته حذف می‌گردد و نشانه بعدی در رشته ورودی به عنوان نشانه جاری در نظر گرفته می‌شود

* تجزیه کننده دو انتقال تطبیق و پیشگو را تا انجام می‌دهد، تجزیه وقتی تمام می‌شود که پشته و رشته ورودی خالی شده باشد.

* مراحل ساخت جدول تجزیه را می‌توان برای هر قاعده تولید $A \to \alpha$ در سه مرحله خلاصه کرد:

۱) برای هر پایانه a (به جز ϵ) در مجموعه first(α)، قانون $A \to \alpha$ به جدول $M[A, a]$ اضافه می‌گردد

۲) اگر ϵ در first(α) باشد برای هر پایانه b در follow(A)، $M[A, b] = A \to \epsilon$ می‌گردد.

۳) در خانه‌های خالی جدول error قرار می‌دهیم.

مثال. جدول تجزیه گرامر زیر را ایجاد کنید

گرامر: $expr \to term\ rest$
$rest \to +expr \mid -expr \mid \epsilon$
$term \to id$

ابتدا تمام first‌های سمت راست قواعد تولید و follow تمام غیرپایانه‌ها را محاسبه می‌کنیم

first(term rest) = {id}
first(± expr) = {±}
first(ϵ) = ϵ
follow(expr) = {$}
follow(term) = {+, -, $}
follow(rest) = {$}

	id	+	-	$
expr	expr → term rest			
term	term → id			
rest		rest → +expr	rest → -expr	rest → ϵ

expr → term rest

جزوه کامپایلر فصل ۳ ص ۲۰   * گرامرهای LL(1) * یکی از ویژگی های مهم گرامرها است.

* اگر در ساختن درخت تجزیه فقط با دیدن یک نشانه بعدی از رشته ورودی در پیمایش چپ به راست، بتوان غیر پایانه بعدی را برای گسترش تشخیص داد، در این صورت گرامر مشتق از آن را LL(1) می نامند و در این صورت سمت چپ ترین اشتقاق ایجاد میشود.

مثال) گرامر ذیل یک گرامر LL(1) است.

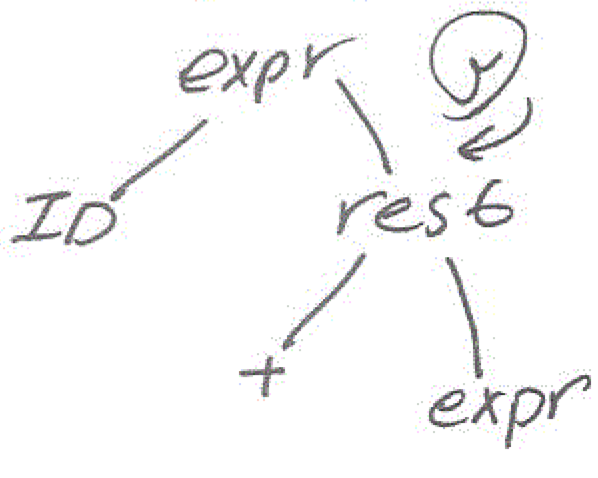
$expr \rightarrow ID\ rest$

$rest \rightarrow '+'\ expr \mid '-'\ expr \mid \epsilon$

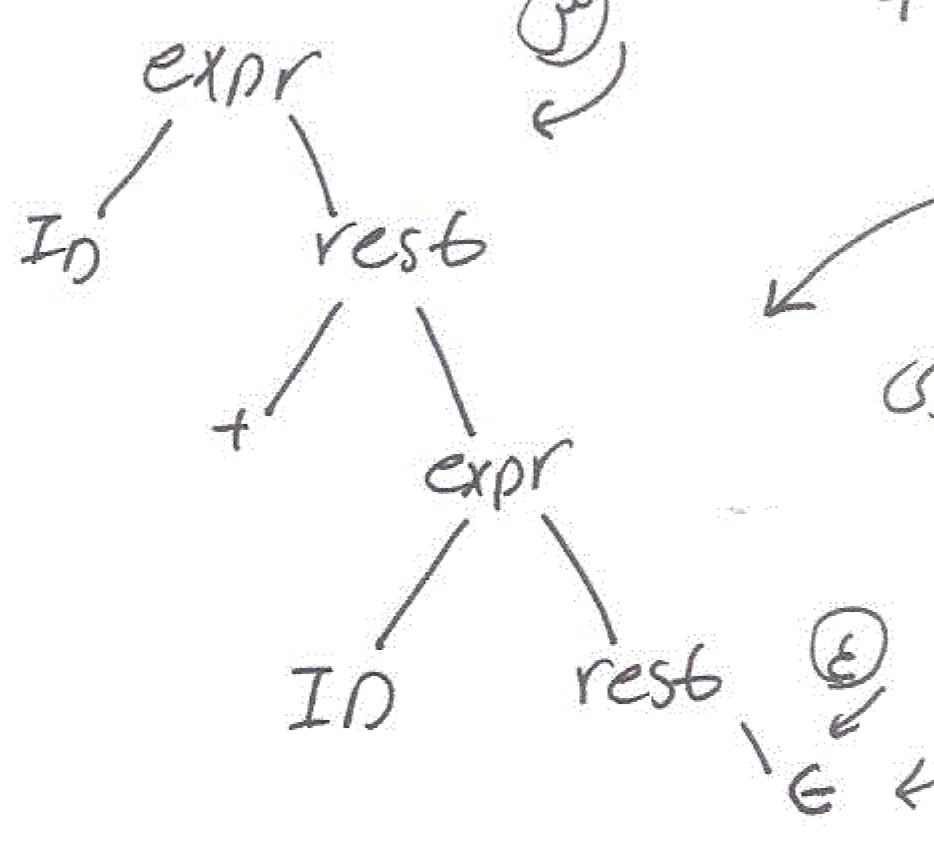
$rest \rightarrow '+'\ expr \mid$ or $'-'\ expr \mid$ or ϵ

به عنوان مثال رشته ID + ID را در نظر می گیریم. اولین نماد ID است. واضح است که در این شرایط تنها انتخاب قاعده تولید مناسب $expr \rightarrow ID\ rest$ است درخت تجزیه زیر شکل می گیرد ←

حال سه انتخاب برای گسترش rest وجود دارد، فقط با توجه به یک نماد بعدی یعنی (+) می توان تشخیص داد که گسترش $rest \rightarrow '+'\ expr$ صحیح است ←



با توجه به ID در ورودی در مرحله بعدی برای گسترش expr فقط یک انتخاب به صورت $expr \rightarrow ID\ rest$ وجود دارد در نتیجه درخت



به دست می آید.

سپس در مرحله آخر ۳ انتخاب برای rest وجود دارد اما چون ورودی تمام شده است $rest \rightarrow \epsilon$ را باید تولید کنید.

مثال) آیا گرامر ذیل LL(2) است؟ ← $S \rightarrow aSb \mid ab$

حل. بله زیرا اگر ۲ نماد رشته ورودی را در نظر بگیریم می توان قاعده تولید بعدی را مشخص کرد، به طور مثال اگر دو نماد بعدی ab باشد $S \rightarrow ab$ انتخاب میشود و اگر aa باشد $S \rightarrow aSb$ انتخاب میشود.

* برای تشخیص گرامرهای LL(1) می توان از قواعد زیر کمک گرفت:

۱) گرامرهای دارای بازگشتی چپ LL(1) نیستند. ۲) گرامرهای مبهم LL(1) نیستند.

۳) اگر جدول تجزیه غیر بازگشتی پیشگو دارای خانه ای با بیش از یک وارده داشته باشد گرامر LL(1) نیست.

۴) اگر در گرامر قاعده تولیدی به صورت $A \rightarrow \alpha \mid \beta$ ($A \rightarrow \alpha$ or β) وجود داشته باشد، به طوری که α و β با یک پایانه شروع شوند گرامر LL(1) نیست به عبارت دیگر اگر $first(\alpha) \cap first(\beta) \neq \emptyset$ قضیه first/first رخ داده و LL(1) نیست.

* مهمترین روش های تست گرامر LL(1) جدول تجزیه می باشد. ۵) اگر شرایط $first(\beta) \cap follow(A) \neq \emptyset$ برقرار باشد آنگاه LL(1) نیست.

خود طراحی کامپایلر فصل ششم ص ۲۱   * گرامرهای LL(1) .   * نکته پایین صفحه را مطالعه نمایید

مثال) گرامر ذیل را در نظر می‌گیریم

$S \to Aab$

$A \to a|\epsilon$

حل. در قاعده تولید $A \to a|\epsilon$ ، $\alpha = a$ و $\beta = \epsilon$ است، با توجه به $S \to Aab$ ، پایانه a بعد از A قرار دارد در نتیجه follow(A)=a و با توجه به $A \to a|\epsilon$ می‌توان دید که first(a)=a است در نتیجه

$first(a) \cap follow(A) = \{a\} \neq \emptyset$

برخورد first/follow رخ می‌دهد و در نتیجه LL(1) نیست.

* ادامه راه‌کار تشخیص گرامرهای LL(1): تا مرحله ۵ را قبلاً گفته شد. ۶) اگر در گرامر، قاعده تولیدی به صورت $A \to \alpha|\beta$ باشد به طوری که α و β هر دو رشته تهی ایجاد کنند، گرامر LL(1) نیست.

* روش‌های تبدیل گرامر غیر LL(1) به LL(1): ۱) حذف بازگشتی چپ ۲) فاکتورگیری چپ

---

① * مثال: آیا گرامر زیر LL(1) است؟

گرامر

$A \to aCbAB|d$

$B \to eA|\epsilon$

$C \to c$

حل ۱. گرامر مبهم است در نتیجه LL(1) نیست.

حل ۲) جدول تجزیه را رسم می‌کنیم

ابتدا first سمت راست تمام غیرپایانه‌ها

$first(aCbAB) = \{a\}$

$first(d) = \{d\}$

$first(eA) = \{e\}$

$first(c) = \{c\}$

قاعده تولید	first سمت راست	وارده جدول تجزیه
$A \to acbAB$	$first = \{a\}$	$M[A,a] = A \to acbAB$
$A \to d$	$first(d) = \{d\}$	$M[A,d] = A \to d$
$B \to eA$	$first(eA) = \{e\}$	$B \to eA$
$C \to c$	$first(c) = \{c\}$	$C \to c$

جدول محاسبه first

غیرپایانه	نماد ورودی					
	d	c	e	a	b	$
A	$A \to d$			$A \to acbAB$		
B			$B \to \epsilon$ $B \to eA$			$B \to \epsilon$
C		$C \to c$				

جدول تجزیه مشکل غیر بازگشتی   دو نوشته

---

② * مثال آیا گرامر زیر LL(1) است؟

$S \to Aa$

$S \to Bb$

$A \to \epsilon$

$B \to \epsilon$

$A \to cAb$

$B \to dAa$

حل. جدول تجزیه را رسم می‌کنیم

S کی میتونه a بشه؟!

غیرپایانه	نماد ورودی				
	a	b	c	d	$
S	$S \to Aa$	$S \to Bb$	$S \to Aa$	$S \to Bb$	
A	$A \to \epsilon$	$A \to \epsilon$	$A \to cAb$		
B		$B \to \epsilon$		$B \to dAa$	

$first(Aa) = \{a, c\}$

$first(Bb) = \{b, d\}$

پس LL(1) هست.

* نکته: برای سهولت در بررسی جدول تجزیه، حروف بزرگ را طبق جدول بالا، غیرپایانه‌ها و حروف کوچک به علاوه $ را در قسمت نمادهای ورودی وارد نمایید حال به طور مثال برای خانه (۱و۱) در جدول بالا که (S و a) هست بگویید "S کی میتونه a بشه" و این جمله را برای تک تک خانه‌ها بگویید و به گرامر نگاه کرده و جواب را یادداشت کنید* الان طراحی * اگر نیاز شد $\alpha \to \epsilon$

با توجه به لمی که در ص قبل ارائه گشته خاطرنشان می‌کنیم این لم با سهولت خوبی جدول تجزیه را برای شما رسم خواهد نمود ولی بعد از رسم جدول می‌بایست قوانین تشخیص دیگر، به غیر از منحصر بودن را نیز در جدول اعمال نمود تا اگر خانه‌ای شامل ۲ وارده می‌باشد مشخص شود. به طور مثال در مثال ① ص قبل که گرامر LL(1) نبود شما با استفاده از پایانه و غیرپایانه و با بررسی جمله "A" که متوجه شده‌اید " و ادامه این جمله برای بقیه ستون‌های جدول، کلیه مؤلفه‌های جدول را یافته و در عرض چند ثانیه بدون محاسبه هیچ کار اضافه‌ای دیگر جدول تجزیه را داشته باشید در این صورت خانه (e و B) در جدول شما دارای مقدار $B \to eA$ خواهد بود حال پس از تکمیل جدول با بررسی قانون ۲ متوجه می‌شوید که خانه (B و e) علاوه بر $B \to eA$ دارای $B \to \varepsilon$ نیز خواهد بود چرا که با توجه به قانون ۲ چون ε در first(B) قرار دارد روی هر پایانه دیگر در follow(B) ← $B \to \varepsilon$ در M[B,e] ظاهر می‌شود.

---

* بررسی خطا در تجزیه کننده: ۱) توقف تجزیه: که باعث طولانی شدن خطایابی و تولید برنامه می‌شود

۲) پوشش خطا: ۲-۱) ترمیم خطای بدون تصحیح: در این روش ورودی تغییر نمی‌کند بلکه تمام اطلاعات تجزیه کننده از بین می‌رود و سپس تجزیه کننده بقیه برنامه را تجزیه می‌کند.

۲-۲) روش تصحیح خطا: در این روش تجزیه کننده جریان ورودی و یا تجزیه کننده را اصلاح می‌کند تا تجزیه ادامه یابد و عبارتند از:

الف) تصحیح حالت اضطراری: در این روش هنگام کشف خطا، نمادهای ورودی را تا رسیدن به یک علامت هماهنگ کننده نادیده می‌گیرد؛

ب): استفاده از قوانین تولید خطا: اگر خطا پیش‌بینی بشود می‌توان قواعد تولیدی به آنها اضافه کرد تا این خطا را [illegible] شود.

مثال) گرامر زیر نشان می‌دهد که بعد از هر دستور سمی‌کالن قرار دارد ← stmt → stmt ; stmt_list | stmt;

با اضافه کردن قاعده تولیدی زیر می‌توان اگر خطایی در صورت درج سمی‌کالن صورت نپذیرد برنامه متوقف نشده و تجزیه کننده با ارسال پیام مناسب به کار خود ادامه دهد ←

stmt error → stmt
stmt_list | stmt

---

* پوشش خطا در تجزیه کننده پیشگوی غیر بازگشتی: همانطور که گفته شد یکی از روش‌های پوشش خطا، تصحیح خطا در حالت اضطراری است که در این روش ورودی تا زمانی که یکی از نمادهای هماهنگ کننده ظاهر گردد، حذف می‌شوند. کارآیی این روش به نحوه تعیین نمادهای هماهنگ کننده بستگی دارد. برخی از روش‌های این انتخاب را در ص بعد مورد بررسی قرار می‌دهیم.

* روش های انتخاب نمادهای هماهنگ کننده *

۱) نماد بالای پشته حذف می شود. در این روش اگر نماد ورودی و بالای پشته یکسان نباشد ابتدا نماد بالای پشته، در ورودی اضافه شده و سپس در مرحله بعد با توجه به این که هر دو نماد بالای پشته و نماد ورودی (اضافه شده) یکی هستند، هر دو حذف و کار تجزیه ادامه می یابد.

۲) نمادهای First(A) را به عنوان عضو مجموعه هماهنگ کننده A در نظر می گیریم. بنابراین اگر یکی از نمادهای First(A) در ورودی ظاهر شود تجزیه براساس A ادامه می یابد. یعنی اگر در پشته به نمادی برخورد کردیم که First نماد بالای پشته و نماد ورودی خالی بود آن نماد را حذف می کنیم و نماد ورودی بعدی را بررسی می کنیم و این کار ادامه می یابد تا M[نماد ورودی، نماد پشته] جزو First شود.

۳) نمادهای Follow(A) را به عنوان نمادهای هماهنگ کننده در نظر می گیریم و مثل روش قبل عمل می کنند. این روش معایبی دارد:

۴) برای حذف معایب روش ۳، کلمات کلیدی شروع کننده دستورات را به مجموعه هماهنگ کننده اضافه می کنیم.

* تجزیه کننده پایین به بالا: از برگ به بالا تا ریشه به پیش می رود و اگر درخت تجزیه ساخته شود رشته پذیرفته خواهد شد.

این تجزیه کننده عکس عمل اشتقاق را انجام می دهد

* تجزیه کننده عملگر - اولویت: به راحتی ساخته می شود - ضعیف است و برای عبارات محاسباتی به کار می رود - از روابط عملگرها برای تعیین دستگیره ها استفاده می کند - نمی توان برای همه گرامرها ایجاد کرد - فقط برای گرامرهای عملگر قابل تولید است که ۲ ویژگی دارند:

۱) سمت راست هیچ قاعده ای ε نباشد ۲) در سمت راست هیچ قاعده ای بیش از یک غیرپایانه مجاور هم وجود نداشته باشد.

* عملگر $a > b$: اولویت a از b بیشتر است، $a < b$: اولویت a از b کمتر است، $a = b$: اولویت a برابر با اولویت b است، عملگرهای یاد شده مثل عملگرهای ریاضی اند با این تفاوت که الزاماً نباید یکی از آنها برقرار باشد ممکن است دو حالت و یا هیچ حالت باشد

* تجزیه کننده های LR: تجزیه کننده های LR ورودی را از چپ به راست پویش و معکوس سمت راست ترین اشتقاق را ایجاد می کنند. پایان رشته ورودی باید علامت خاص داشته باشد تا تجزیه کننده LR با رسیدن به آن، پایان رشته را تشخیص دهد. این علامت نباید از پایانه های گرامر باشد. این تجزیه کننده دستگیره ها را کشف و سپس کاهش می دهند و برای کشف دستگیره از پشته استفاده می کنند.

* قسمتهای مختلف این تجزیه کننده: ۱) ورودی (به انتهای رشته ورودی $ اضافه میشود) ۲) پشته به صورت $S_0X_1S_1...X_mS_m$ که S_i نشان دهنده حالات و X_i پایانه ها و غیرپایانه های گرامر است، نمادهای ورودی به پشته منتقل می شوند تا یک دستگیره در پشته یافته شود ۳) جدول تجزیه شامل دو قسمت goto و Action است که به ترتیب به معنی حالت بعدی و عملی که باید انجام شود است.

۴) خروجی ۵) برنامه تجزیه کننده LR: قسمت اصلی تجزیه کننده است، اساس نماد جاری، حالت بالای پشته، محتوای جدول و مرحله بعدی را تعیین می کند.

 * تجزیه کننده‌های LR

* برنامه تجزیه LR با توجه به محتوای جدول اعمال ذیل را انجام می‌دهد: ۱) انتقال، ۲) کاهش، ۳) پذیرش ۴) خطا

* ۱) انتقال: اگر $action[i,a]=sm$ باشد، ابتدا نماد a و سپس m را به بالای پشته منتقل می‌کند.

* ۲) کاهش: اگر $action[i,a]=rm$ باشد، یک دستگیره یافته شده است که باید کاهش یابد. اگر m شماره قاعده $A \to \beta$ باشد دنباله β از پشته حذف می‌شود. حال اگر بالای پشته m باشد ابتدا A و سپس عدد موجود در $goto[m,A]$ را به بالای پشته اضافه می‌کند. لازم به ذکر است که برای حذف β از بالای پشته تعداد $2|\beta|$ (که $|\beta|$ تعداد نمادها یا طول β است) از بالای پشته حذف می‌شوند.

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

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

* روش‌های ساخت جدول تجزیه: ۱) LR(0): ساده‌ترین و ضعیف‌ترین روش ← LR(0)

۲) SLR(1): (LR ساده): این روش از LR(0) قوی‌تر است ← SLR(1)

۳) LR(1) یا LR متعارف: قوی‌ترین روش ساخت جدول تجزیه LR است. ← LR(1)

۴) LALR(1): از SLR(1) قوی‌تر و از LR(1) ضعیف‌تر است. ← LALR(1)

---

* روش LR(0): عنصر LR یک قاعده تولید است که نقطه‌ای در سمت راست آن قرار دارد. شکل $S \to \alpha.\beta$، مکان نقطه در عنصر LR میزان پیشرفت، در کاهش غیرپایانه سمت چپ (یا همان S) را نشان می‌دهد.

* عنصر $S \to \circ\alpha\beta$ نشان می‌دهد که برای کاهش S نیاز به کاهش α است زیرا علامت نقطه قبل از α است و پیشرفتی در آن نداشتیم. اگر α را کاهش دهیم این پیشرفت را با عنصر $S \to \alpha.\beta$ نمایش می‌دهیم و یعنی α کاهش یافته و حالا قصد کاهش β را داریم. اگر β را کاهش دهیم عنصر مشابه ← $S \to \alpha\beta\circ$ پس از کاهش β می‌توان $\alpha\beta$ را به S کاهش داد. عنصر $S \to \alpha\beta\circ$ نشان می‌دهد که یک دستگیره که همان $\alpha\beta$ است یافت شده است. انواع عناصر عبارتند از:

۱) عنصر کاهشی: عنصری که نقطه در آخر سمت راست قاعده تولید قرار دارد که نشان دهنده یک دستگیره است. $S \to XYZ\circ$ (کاهش دستگیره یافته)

۲) عنصر انتقالی: عنصری که کاهشی نباشد $S \to X.YZ$ یک عنصر انتقالی است.

* عنصر انتقالی $S \to \alpha.a\beta$ (a پایانه) می‌گویند اگر a در نماد جاری ورودی باشد از ورودی حذف و $S \to \alpha a.\beta$ می‌شود.

* عنصر انتقالی $S \to \alpha.X\beta$ (X غیرپایانه) می‌گویند که دستگیره‌ای از X باید به دست آید یا کاهش X عنصر به ← $S \to \alpha X.\beta$ تبدیل می‌شود.

جزوه کامپایلر فصل ششم ص ۲۵ *تجزیه‌کننده‌های LR

گرامر روبرو را در نظر بگیرید ← ابتدا قاعده تولید $S' \to E$ را به آن اضافه می‌کنیم

①

$E \to E + T$

$E \to T$

$T \to id$

②

$S' \to E$

$E \to E + T$

$E \to T$

$T \to id$

اولین عنصر $S' \to .E$ است که فرض‌های ما در مورد رشته ورودی بیان می‌کند. علامت نقطه قبل از E نشان می‌دهد که ممکن است رشته‌ای از E به دست آید. با توجه به $E \to E+T$ و $E \to T$، E می‌تواند E+T یا T باشد پس می‌توان آنها را به E کاهش داد. نقطه قبل آنها هم قرار می‌گیرد. مجموعه به شکل ←

$S' \to .E$

$E \to .E+T$

$E \to .T$

با توجه به اینکه $T \to id$ را تولید می‌کند ممکن است از رشته T به دست آید لذا $T \to .id$ را اضافه می‌کنند

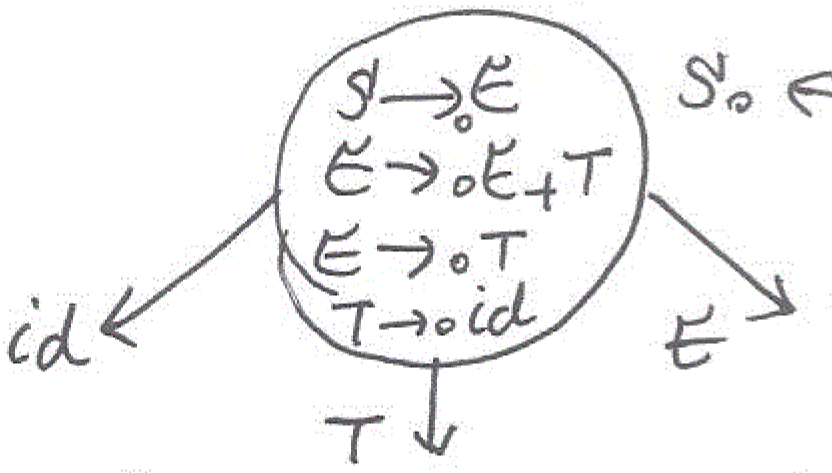
* مجموعه به دست آمده را S_0 می‌نامند. در ادامه روش دیگری را برای به دست آوردن مجموعه S_0 می‌گوییم

* مجموعه عناصر به دست آمده از عنصر $S' \to .E$ را $Closure(\{S' \to .E\})$ می‌نامند و از ۲ قانون ذیل تا زمان اضافه کردن عنصر جدید به مجموعه استفاده می‌کنیم. ۱) I به $Closure(I)$ اضافه می‌گردد

۲) اگر $A \to X.YZ$ در $Closure(I)$ و قاعده تولید $Y \to \alpha\beta$ در گرامر باشد آنگاه $Y \to .\alpha\beta$ به $Closure(I)$ اضافه می‌شود

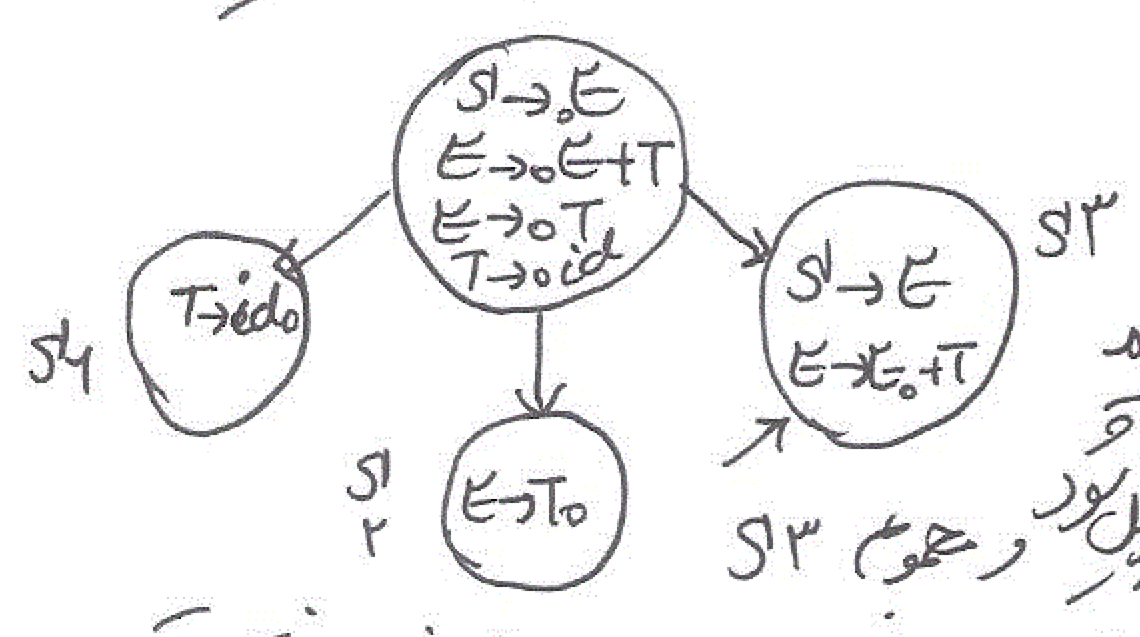
حال ماشین خودکار گرامر یاد شده را رسم می‌کنیم. اولین را مجموعه S_0 ← S_0

و پایه‌ای از نمادهایی که سمت چپ آنها . است قرار می‌دهیم



در ذیل به بررسی سه پایه id و T و E در مجموعه S_0 می‌پردازیم. id: نماد id باعث می‌شود علامت نقطه در عنصر $T \to .id$ پیشروی کرده و از id عبور کند در این صورت $T \to id.$ تک عنصر کاهشی است که نشان دهنده پایان رشته است

چون سمت راست نقطه نماد دیگری وجود ندارد این مجموعه را S'_1 می‌نامیم

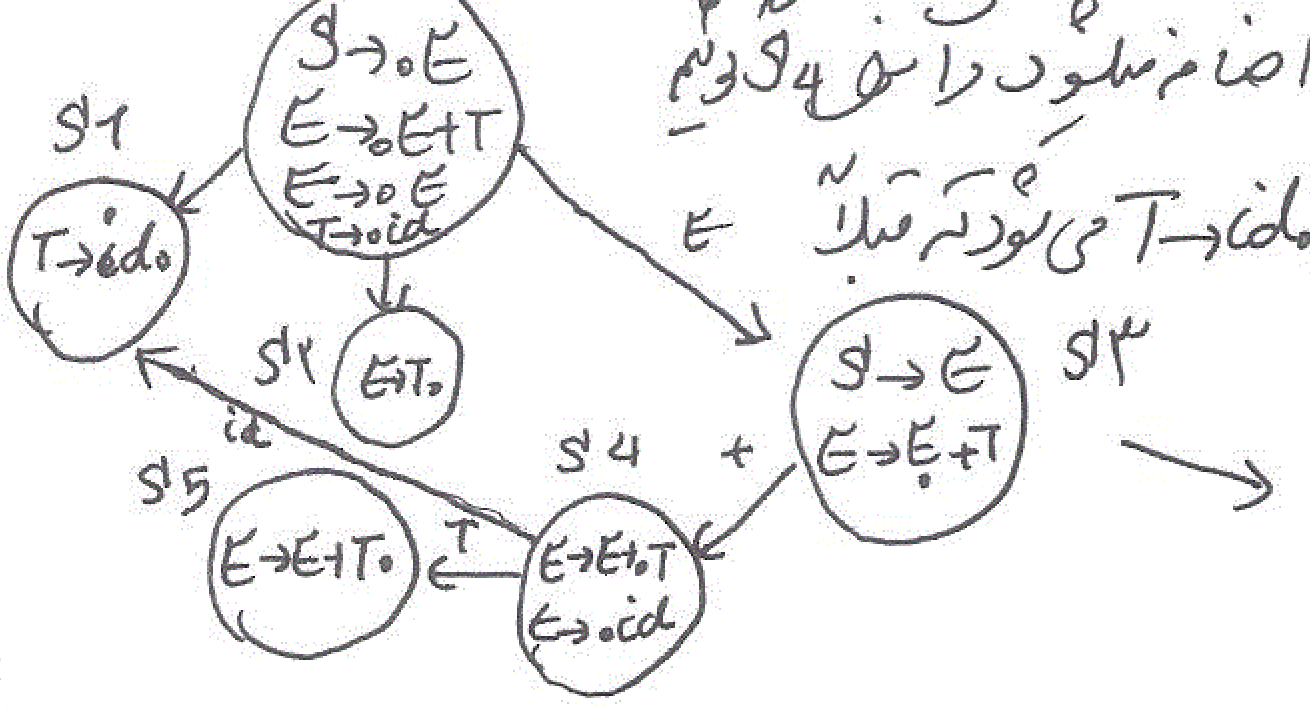


۲) نماد T باعث می‌شود $E \to T$ به $E \to T.$ تبدیل شود این مجموعه را S'_2 می‌نامیم

۳) نماد E باعث می‌شود $E \to E+T$ و $S' \to E$ به ترتیب به $E \to E.+T$ و $S' \to E.$ تبدیل شود و مجموعه S'_3

حال اگر S'_3 را گسترش دهیم نماد + باعث می‌شود $E \to E+.T$ تبدیل شود که با استفاده از این عنصر فرض پایان یک رشته از T را نشان می‌شود و با توجه به $T \to id$، عنصر $T \to .id$ به مجموعه اضافه می‌شود و آن را S'_4 می‌گوییم

* حال عنصر $E \to E+.T$ به $E \to E+T.$ تبدیل و مجموعه S'_5 ایجاد می‌شود و ما $T \to id.$ می‌شود که قبلاً در S'_1 بوده و لذا مجموعه جدید ایجاد نمی‌شود.

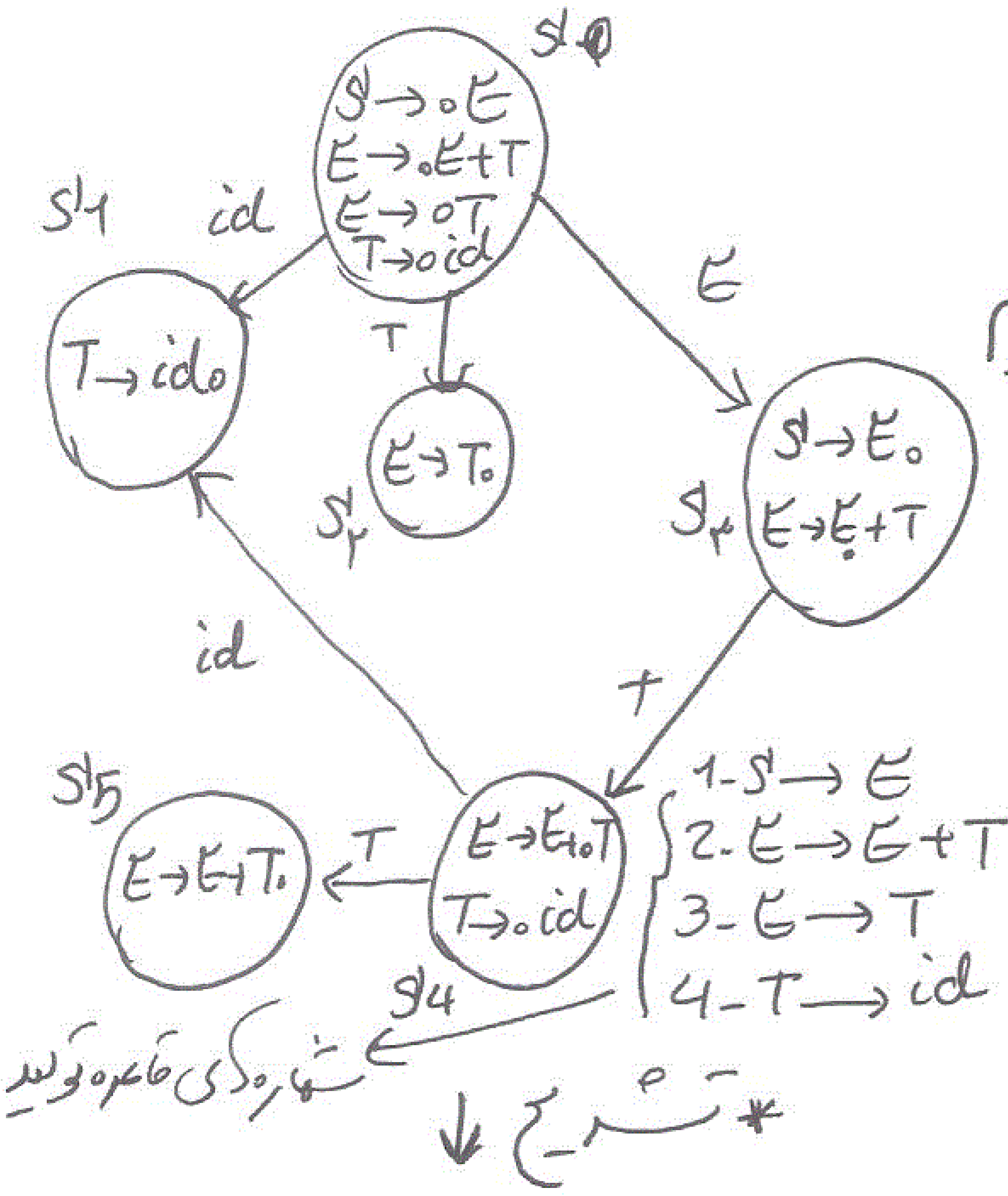


ماشین خودکار LR(0)

خودآموز کامپایلر فصل ششم ص ۲۶

* تجزیه کنندگی LR * L(0)

* با توجه به ماشین خودکار ایجاد شده در ص قبل اکنون جدول تجزیه را ترسیم می‌نماییم



حالت	action (پایانه‌ای)			goto (غیرپایانه‌ای)	
	id	+	$	T	E
0	s1	error	error	2	3
1	r4	r4	r4		
2	r3	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5	r2	r2	r2		

جدول L(0)

* تشریح ↓

* با توجه به ماشین خودکار، همه عناصر را بررسی می‌کنیم ابتدا نماد id باعث تغییر حالت از s0 به s1 می‌شود و عنصر انتقالی T→•id به T→id• تبدیل می‌شود این عمل همراه با انتقال id به پشته است در نتیجه action[0,id] برابر با shift 1 است shift به معنی انتقال id به پشته است و عدد ۱ نشان می‌دهد که بعد از انتقال تجزیه کننده به حالت s1 می‌رود به منظور اختصار به جای shift 1 از s1 استفاده می‌کنیم ← action[0,id]=s1 ، چون با نمادهای + و $ از s0 به خروجی وجود ندارد اگر action این حالت اتفاق بیفتد، خطا رخ داده است.

نمادهای E و T نیز باعث تغییر حالت از s0 به s2 و s3 می‌شوند و چون غیرپایانه هستند ← goto[0,E]=s3 و goto[0,T]=s2

* حالت s1: این حالت شامل عنصر کاهشی T→id• است در این حالت تکه دستگیره یافت شده است در نتیجه به قاعده تولید T→id کاهش انجام می‌شود بنابراین در سطر ۱ جدول reduse T→id قرار می‌گیرد به منظور اختصار از r و از شماره قاعده تولید به جای قاعده تولید استفاده نموده‌ایم چون عمل کاهش صرف نظر از نماد ورودی انجام می‌شود برای همه پایانه‌ها r4 وارد می‌کنیم

یعنی ← action[1,id],[1,+],[1,$]=r4

* حالت s2: این حالت شامل عنصر کاهشی E→T• است و در این حالت تکه دستگیره یافت شده است ولذا r3 برای همه پایانه‌ها

* حالت s3: حالت s3 شامل عنصر کاهشی S'→E• است اگر نماد ورودی $ باشد رشته ورودی تمام شده ← action[3,$]=accept

نماد + باعث تغییر حالت به s4 می‌شود و نماد id در حالت s3 بی اثر است ← action[3,+]=s4, action[3,id]=error

* حالت s4: همه عناصر در این s4 انتقالی هستند در نتیجه نمادهای T و id که بلافاصله بعد از نقطه قرار دارند باعث تغییر حالت می‌شوند و نتایج در جدول ثبت می‌شود. action[4,id]=s1 و goto[4,T]=5

* حالت s5: شامل عنصر کاهشی E→E+T• است و دستگیره یافت شده است در نتیجه با توجه به E→E+T کاهش انجام می‌شود سطر ۵ جدول reduse E→E+T یا همون r2 برای تمام پایانه‌ها قرار می‌گیرد * دیدید چه باحال بود؟! :)

* تجزیه کننده‌های LR * تجزیه روش SLR(1)

* روش LR(0) که در بخش قبل مورد بررسی قرار گرفت، بسیار ضعیف است و در بسیاری از کامپایلرها قابل استفاده نیست و علت ضعف آن عدم توجه به نمادهای رشته ورودی در هنگام کاهش تک عنصر کاهشی است و روی همه پایانه‌ها اقدام به کاهش می‌کند. روش هوشمندانه‌تر SLR(1) عنصر کاهشی $N \to \alpha.$ را وقتی به N کاهش می‌دهد که نماد جاری ورودی در follow(N) باشد. در این روش نیز ابتدا ماشین خودکار را رسم می‌کنیم. تفاوت SLR(1) و LR(0) در نحوه تولید جدول تجزیه آن است. با توجه به مثال ص قبل این بار یک جدول برای SLR(1) را رسم می‌کنیم.

حالات	action			goto	
		پایانه		غیر پایانه	
	id	+	$	T	E
0	s1	error	error	2	3
1	error	r4	r4		
2	error	r3	r3		
3	error	s4	accept		
4	s1	error	error	5	
5	error	r2	r2		

* جدول SLR(1)

1- $S \to E$
2- $E \to E + T$
3- $E \to T$
4- $T \to id$

شرح ↓

* نماد id باعث تغییر حالت به S_1 می‌شود و عنصر انتقالی به $T \to id.$ تبدیل می‌شود ← action[0,id]=s1

* نمادهای E و T نیز به ترتیب باعث انتقال به ۳ و ۲ می‌شوند و لذا ← goto[0,T]=2 , goto[0,E]=3

* حالت S_1: شامل عنصر کاهشی $T \to id.$ است و در شکل بالا نیز رسم شده است. حال در چه شرایطی کاهش انجام می‌شود؟ برای follow(T) کاهش را انجام است یعنی follow(T)={$,+} و r4 در دو خانه $ و + قرار می‌گیرد و id، error می‌شود.

* حالت S_2: شامل عنصر کاهشی $E \to T.$ است و در شکل بالا نیز رسم شده است یعنی follow(E)={+,$} لذا r3 در این خانه‌ها قرار می‌گیرد. در اصل در حالت دوم action[2,+]=r $E \to T$ و action[2,$]=r $E \to T$ می‌باشد.

* حالت S_3: شامل عنصر کاهشی $S \to E.$ در نتیجه نماد شروع S تولید شده است، اگر نماد ورودی $ باشد، رشته ورودی نیز تمام شده است در نتیجه رشته ورودی پذیرفته می‌شود ← action[3,$]=accept همچنین $E \to E.+T$ عنصر انتقالی بوده و باعث تغییر به حالت S_4 می‌شود و عنصر $E \to E+.T$ تولید می‌گردد ← action[3,+]=s4 و id نیز error می‌دهد.

* حالت S_4: همه عناصر در حالت انتقالی هستند، نتایج را در جدول ثبت می‌کنیم.

* حالت S_5: شامل عنصر کاهشی $E \to E+T.$ است و در شکل بالا نیز رسم شده است. follow (E) را محاسبه می‌کنیم. follow (E) قبلاً محاسبه شده و {+,$} می‌باشد لذا همانها را با استفاده از شماره قاعده در جدول مشخص می‌کنیم.

 * گرامر SLR(1) و LR(0) ها و روش LR(1)

* گرامرهایی که برای آنها می توان جدول SLR(1) ایجاد کرد گرامر SLR(1) می نامیم

* مقایسه جداول LR(0) و SLR(1)، با توجه به مقایسه دو جدول ص ۲۶ و ص ۲۷ که مربوط به این دو گرامر هستند در می یابیم که در LR(0) ها کلیه خانه های یک سطر دارای عنصر کاهشی بوده، کاهش را تواماً در SLR(1) این کار توسط follow ها صورت پذیرفته است

* روش LR(1): کاهش یک عنصر کاهش مانند A→α در صورتی مجاز است که نماد ورودی در مجموعه follow(A) موجود باشد، اگر به خاطر داشته باشید این تکنیک در SLR(1) مورد استفاده قرار می گیرد که دقیق نیست زیرا یک غیر پایانه ممکن است شامل چند قاعده تولید به شکل $A \to \alpha_1 | \alpha_2 | \dots | \alpha_n$ باشد و یک کاهش نادرست صورت گیرد برای حل این شکل باید هر عنصری که پایانه های مورد انتظار را با خود همراه داشته باشد که این پایانه های همراه را نماد پیشگو گوییم * روش LR(1) نماد بعدی را پیشگویی می کند، اگر هنگام کاهش نماد جاری ورودی با نماد پیشگویی شده برابر باشد کاهش انجام میشود در غیر اینصورت کاهش انجام نمی شود

* در روش LR(0) هیچ نمادی پیشگویی نمیشود در SLR(1) یک نماد ورودی برای مجاز بودن کاهش استفاده میشود و در LR(1) از یک نماد پیشگو استفاده می گردد، به همین جهت آن را LR(1) نامیدند، روش هایی که بیش از یک نماد استفاده می کنند LR(K) هستند.

* در روش LR(1) نیز ابتدا عناصر را تولید و سپس ماشین خودکار را برای آنها درست می کنیم، به طور مثال گرامر ذیل را در نظر بگیرید:

$S \to E \mid ab$
$E \to dEb \mid T$
$T \to a$

$\Longrightarrow$

1- $A \to S$
2- $S \to E$
3- $S \to ab$
4- $E \to dEb$
5- $E \to T$
6- $T \to a$

(1) $[A \to .S, \$]$

* اولین نماد پیشگویی $[A \to .S, \$]$ است و بیان می کند کاهش A به S مجاز است که نماد ورودی $ باشد، به عبارت دیگر بعد از S نماد $ قرار داشته باشد، در اینجا نقطه قبل از S قرار دارد بنابراین ابتدا باید S تولید گردد برای تولید S

دو راه ۱) کاهش ab به S و یا کاهش E به S وجود دارد، با توجه به اینکه این دو هنوز گسترش نیافته اند نقطه قبل از آنها قرار می گیرد
همچنین کاهش ab و E به S در حالتی مجاز است که نماد ورودی $ باشد لذا (2) $[S \to .ab, \$]$ و (3) $[S \to .E, \$]$

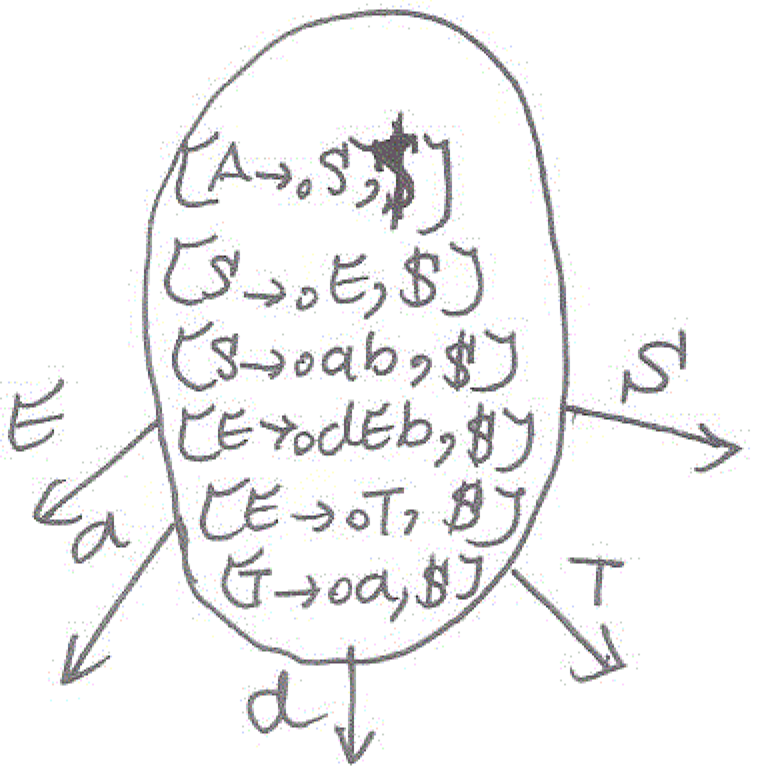
با استفاده از پیشگویی $[S \to .E, \$]$ می توان پیشگویی های دیگری استنتاج نمود ← (4) $[E \to .T, \$]$ و (5) $[E \to .dEb, \$]$

با استفاده از پیشگویی های بدست آمده سعی می کنیم پیشگویی های دیگری انجام دهیم ← (6) $[T \to .a, \$]$

مجموعه بدست آمده را I_0 می نامیم که یکی از حالات ماشین خودکار است، علائم T, d, a, E که سمت راست نقطه قرار دارند نشان دهنده انواع انتقال های ممکن از I_0 هستند، نتیجه این بررسی را در شکل ص بعد مشاهده کنید

جزوه کامپایلر فصل سوم ص ۲۹

* تجزیه کننده‌های LR * روش LR(1)



1- A → S
2- S → E
3- S → ab
4- E → dEb
5- E → T
6- T → a

* حال به تشریح کامل این گرامر می‌پردازیم:

* نماد E باعث تبدیل [S→E., $] می‌شود و کاهش است آن را S1 می‌نامید

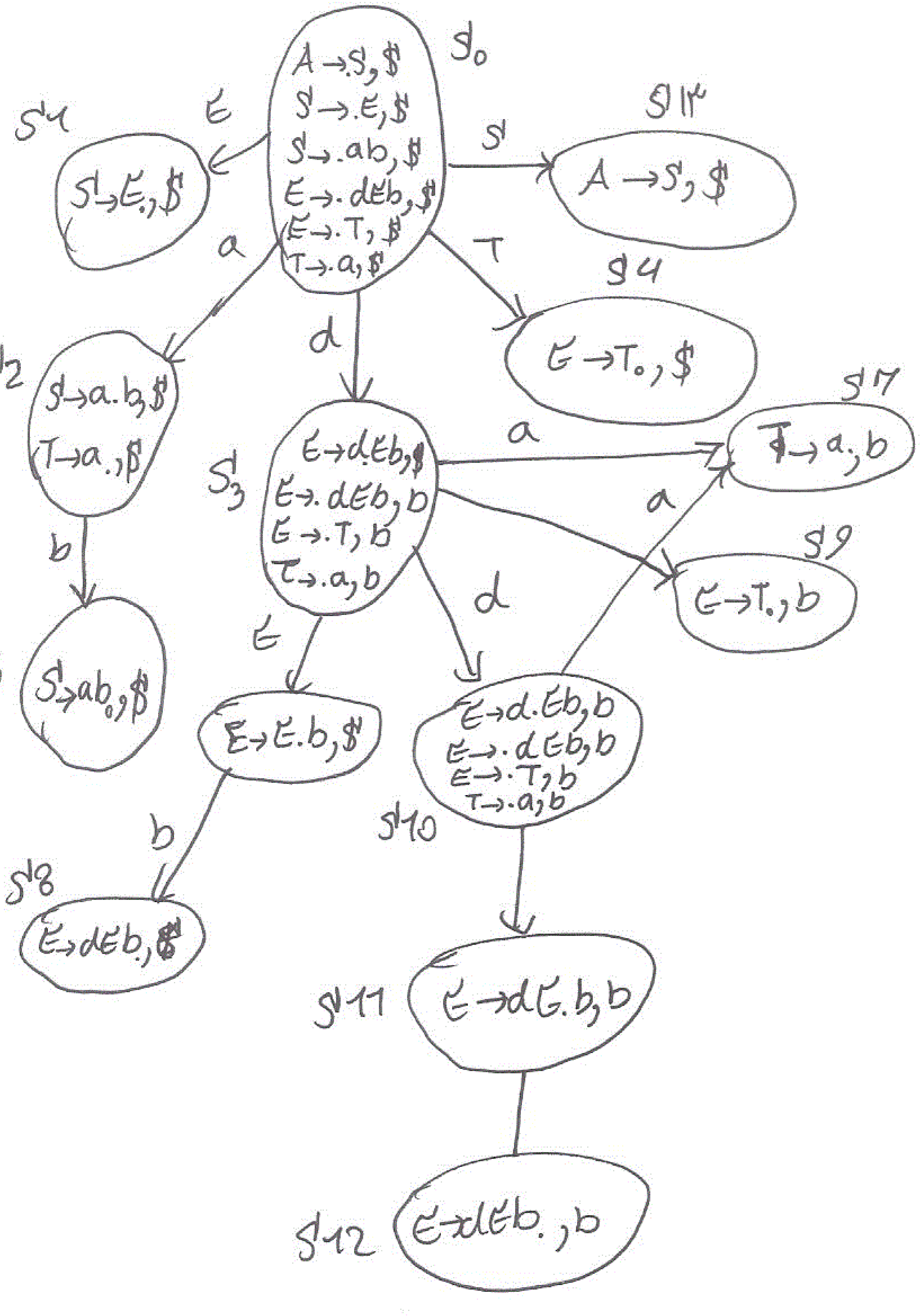
نماد a باعث تبدیل [T→a., $] می‌شود و کاهش است همچنین

* نماد a در عنصر تبدیل [S→a.b, $] وجود دارد، در نتیجه مجموعه عناصر [T→a., $] و [S→a.b, $] مجموعه S2 را بوجود می‌آورند.

مجموعه S2 یک عنصر کاهش و یک عنصر انتقالی دارد در روش SLR(1) این حالت باعث برخورد انتقال/کاهش می‌گردید ولی در این روش این اتفاق نمی‌افتد. زیرا در عنصر کاهشی کاهش زمانی است که عنصر ورودی $ و در عنصر انتقالی انتقال زمانی است که نماد ورودی b باشد.

* نماد d باعث می‌شود تبدیل [E→d.Eb, $] صورت پذیرد و پیشگویی‌های دیگر را می‌یابید و لذا [E→.T, b] بوجود می‌آید و همچنین با توجه به اینکه بلافاصله بعد از E نماد b قرار دارد هم در مجموعه [E→.T, b] و هم در [E→.dEb, b]، b باید نماد ورودی باشد عنصر [E→.T, b] نیز با توجه به T→a، [T→.a, b] پس از کاهش تبدیل می‌شوند و مجموعه‌ای که نماد d را S3 می‌نامید

با ادامه این روند شکل زیر حاصل می‌شود



حالت	action				goto		
*	a	b	d	$	E	T	S
0	s2		s3		1	4	13
1	r2			r2			
2		s5		r6			
3	s7		s10		6	9	
4				r5			
5				r3			
6		s8					
7		r6					
8				r4			
9		r5					
10	s7		s10		11	9	
11		s12					
12		r4					
13				accept			
—	—	—	—	—	—	—	—

خود. کامپایلر فصل ششم ص ۳۰ * تجزیه کننده * روش LALR(1) از LR(0) و SLR(1) قوی‌تر از LR(1) ضعیف‌تر

* روش‌های LR(0) و SLR(1) ضعیف هستند و روش LR(1) به اندازه کافی قوی هستند. اما حالات LR(1) خیلی زیاد است که خود مشکل بزرگی برای ساخت جدول تجزیه است. تکنیک LALR(1) راه حل میانه است، تعداد حالات کمتری نسبت به LR(1) ایجاد می‌کند به همین جهت استفاده از آن، برای زبان‌های برنامه‌نویسی عملی‌تر است.

* برای تشریح LALR(1) ابتدا حالات هسته را تعریف می‌کنیم. حالت هسته مجموعه‌ای از عناصر پیش‌گویی است که پایانه‌های پیش‌گویی آن حذف شده است، به عنوان مثال حالت s3 مثال قبل که در ص ۲۹ قبل و توسط روش LR(1) تولید شده بود را در نظر می‌گیریم

حالت s3 از پیش‌گویی LR(1)	حالت هسته LR(1)
E → d.Eb , $	E → d.Eb
E → .dEb , b	E → .dEb
E → .T , b	E → .T
T → .a , b	T → .a

* می‌بینیم که هسته‌های مربوط به حالت‌های LR(1) معادل حالت‌های LR(0) هستند در واقع LR(1) هر یک از حالت‌های LR(0) را به چند حالت شکسته و این شکستن باعث قدرت LR(1) هست ولی در برخی موارد این شکستن که نیازی ندارد انجام شود

* ماشین خودکار LALR(1) شامل دو دسته از حالات است که عبارتند از ۱) حالات غیر ترکیبی ۲) حالات ترکیبی

* ۱) حالات غیر ترکیبی: حالاتی هستند از ماشین خودکار LR(1) که ترکیب نشده‌اند مثل s0, s1, s2, s5, s7, s13

* ۲) حالات ترکیبی: حالاتی که از ترکیب حالات LR(1) به دست آمده‌اند مثل s3,10 که ترکیب حالات s3 و s10

* با توجه به تغییرات یاد شده جدول تجزیه به شکل زیر در می‌آید.

LALR(1)	LR(1)
از s0: s3,10	از s0: s3
از s3,10: s6,11	از s3: s6
از s3,10: s7	از s3: s7

تناظر تغییرات

جدول تجزیه و پذیرش رشته dab$

حالات	action				goto		
	a	b	d	$	E	T	S
0	s2		s3,10		1	4,9	13
1	r2			r2			
2		s5					
3,10	s7		s3,10		6,11	4,9	
4,9		r5		r5			
5				r3			
6,11		s8,12					
7		r6					
8,12				r4			
13				accept			
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—

حالت‌های حذف شده

پذیرش رشته dab$

پشته	رشته ورودی
0	dab$
0d3,10	ab$
0d3,10a7	b$
0d3,10a7	b$
0d3,10T	b$
0d3,10T4,9	b$
0d3,10T4,9	b$
0d3,10E	b$
0d3,10E6,11	b$
0d3,10E6,11b8,12	$
0E	$
0E1	$
0S	$
0S13	accept

0d3,10E6,11,b8,12

جزوه کامپایلر فصل ششم ص ۳۱

* گرامرهای مبهم LR نیستند

* علی رغم قدرت زیاد تجزیه کننده‌های LR برای گرامرهای مبهم نمی‌توان تجزیه کننده LR ایجاد کرد. دو راه کار وجود دارد:

۱) تبدیل گرامر مبهم به گرامر غیر مبهم معادل آن   ۲) استفاده از قوانین اضافی

* دو برخورد انتقال/کاهش و برخورد کاهش/کاهش علت شکست در تولید تجزیه کننده LR است. اگر به نحوی این برخوردها را رفع کنند می‌توان تجزیه کننده LR را ساخت. برخورد انتقال/کاهش زمانی رخ می‌دهد که تجزیه کننده در یک حالت خاص امکان انتقال و کاهش را داشته باشد. با استفاده از روش‌های زیر می‌توان این موارد را رفع نمود:

۱) رفع برخورد انتقال/کاهش: سعی می‌کنیم طولانی‌ترین دنباله را بیابیم لذا عمل انتقال انجام می‌گردد. روش دیگر استفاده از تقدم و شرکت پذیری است.

۲) رفع برخورد کاهش/کاهش: قاعده تولیدی انتخاب می‌شود که طولانی‌ترین رشته را کاهش دهد. اگر چند قاعده باشد از اولویت‌بندی استفاده می‌شود. در این صورت گرامری که اول نوشته شده است اولویت دارد.

1- $S' \to E$

2- $E \to E * E$

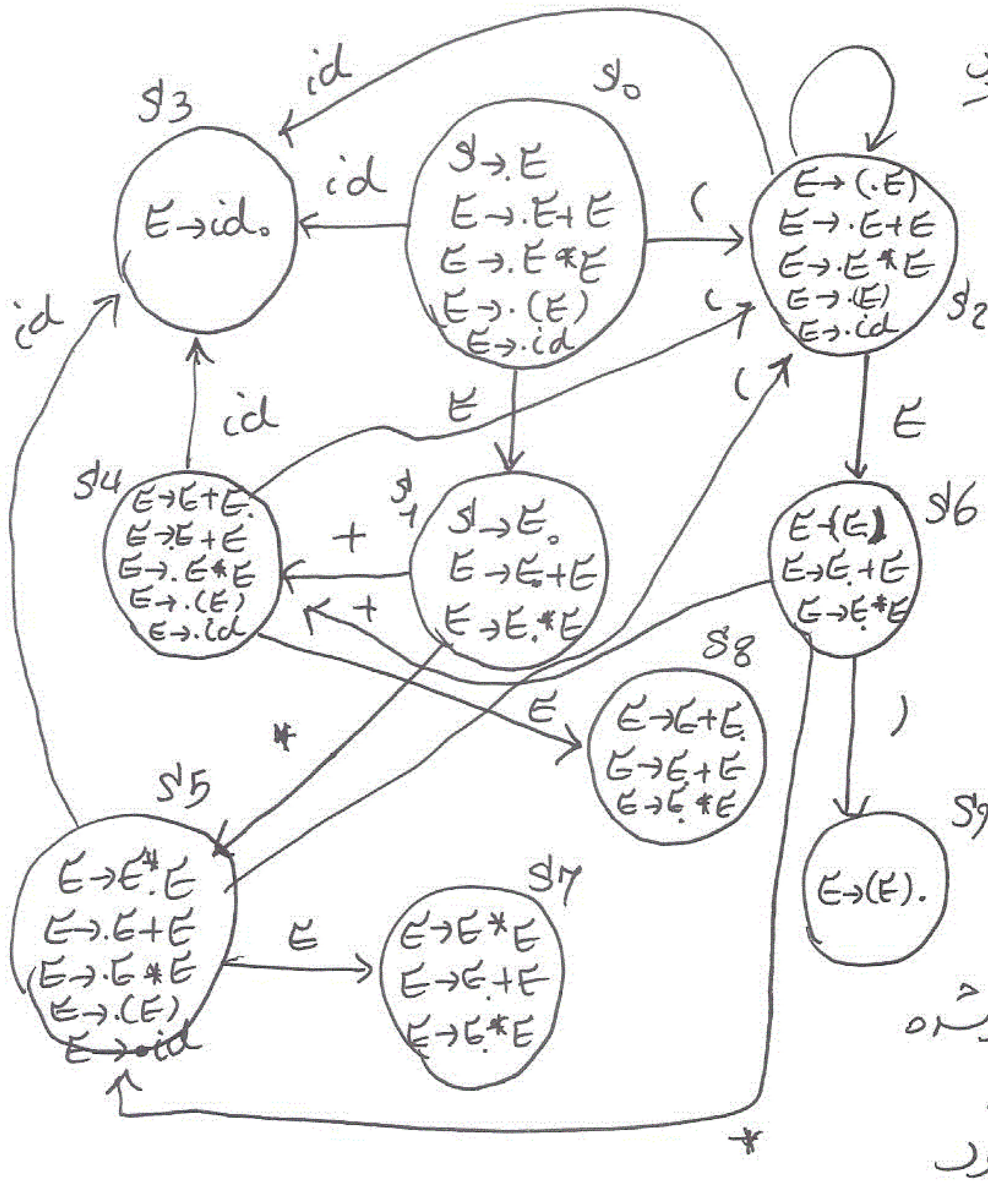
3- $E \to E + E$

4- $E \to (E)$

5- $E \to id$

مثال: گرامر زیر را در نظر بگیرید $E \to E+E \mid E*E \mid (E) \mid id$ ، قواعد آن به صورت زیر درمی‌آید

برای گرامر فوق ماشین زیر تولید می‌شود



رفع برخورد:

۱) در حالت S8 برای رفع برخورد انتقال/کاهش انتخاب می‌گردد

۲) حالت S7 اگر نماد ورودی * باشد، انتقال و اگر + باشد کاهش انتخاب می‌گردد

حال می‌توان با توجه به رفع برخوردهای یاد شده جدول تجزیه این گرامر را ترسیم نمود

جزوه کامپایلر فصل ششم ص ۳۲ * ترجمه

* کامپایلر یک رشته ورودی را به رشته خروجی ترجمه می کند و این فرایند می تواند در ضمن ترجمه انجام شود، تجزیه یک روند مرحله به مرحله است که اگر متناسب با هر مرحله از فرایند تجزیه، ترجمه مورد نیاز انجام شود ترجمه پس از اتمام تجزیه به دست می آید.

* پیاده سازی ترجمه، اگر ترجمه ضمن تجزیه انجام میشود، به همین سادگی است اما باید برای یک گرامر تجزیه کننده را ساخته و سپس دستورات ترجمه متناسب را در تجزیه کننده قرار داد.

* برای ساخته تحلیل گر نحوی روش های مختلفی وجود دارد که برخی از آنها عبارتند از: ۱) ساخته تحلیل گر نحوی با استفاده از زبانهای برنامه سازی سطح بالا مثل C و پاسکال ۲) استفاده از ابزارهای تولید کننده تجزیه کننده برای تحلیل گر نحوی مثل yacc در unix، bison در windows

* Bison برای گرامر یک تجزیه کننده پائین به بالا به روش LALR(1) تولید می کند. کامپایلر نویس مشخصات گرامر زبان را با استفاده از نشانه گذاری مستقل از زبان در یک فایل متنی با پسوند y. بیان می کند و به زبان C ترجمه می گردد و در نهایت برنامه به زبان C با استفاده از کامپایلر C، برنامه اجرائی که تجزیه کننده زبان مبدأ است، تولید می گردد.

* فایل مشخص کننده ساختار نحوی زبان مبدأ شامل سه قسمت ۱) اعلانها ۲) قوانین ترجمه ۳) توابع هستند

* Bison تحلیل گر نحوی را در تابعی به نام yyparse() پیاده سازی می کند که این تابع نیز تابع yylex() را فراخوانی می کند که پیاده سازی این تابع یعنی yylex() بر عهده برنامه نویس است و مهمترین تابع در قسمت پیاده سازی همین تابع است. این تابع نشانه را به وسیله دستور return به yyparse() برمی گرداند. برای ارسال مقدار نشانه از متغیر مشترک yylval استفاده میشود

* برنامه نویس تابع main را نیز پیاده سازی می کند که مهمترین دستور در آن فراخوانی yyparse() است. همچنین yyerror را برنامه نویس می نویسد که تابعی است که در صورت بروز خطا توسط yyparse فراخوانی میشود

* برای ترجمه ورودی به خروجی از قوانین ترجمه استفاده می کنیم. درون قوانین ترجمه برای هر پایانه و غیرپایانه (در قاعده تولید) متغیری وجود دارد که حاوی لغت منطبق شده برای غیرپایانه و یا پایانه است. این متغیر قابل دسترسی است و می توان از مقدار آن استفاده کرد و یا تغییر داد.

* پایان جزوه طراحی کامپایلر ۲۳ / ۱۰ / ۹۱

۱۰:٤۰ pm

AC۷۲۷