به نام خدا

طراحی کامپایلر

آرش شفیعی

تحلیل نحوی

تحلیل نحوی

- در این فصل در مورد الگوریتم‌های مختلف تجزیهٔ گرامرها صحبت خواهیم کرد که معمولاً در کامپایلر استفاده می‌شوند.
- ساختار هر زبان برنامه نویسی توسط قوانینی تعیین می‌شود. برای مثال در زبان سی، یک برنامه از تعدادی توابع تشکیل شده است و هر تابع از تعدادی دستورات تشکیل شده که این دستورات می‌توانند تعریف و اعلام متغیرها، انتساب مقدار، دستورات شرطی و حلقه‌های تکرار باشند.
- ساختار نحوی [1] یک زبان، نحوهٔ قرارگیری توکن‌ها در جملات زبان را تعیین می‌کند.
- ساختار نحوی یک زبان را می‌توان توسط گرامرهای مستقل از متن توصیف کرد.

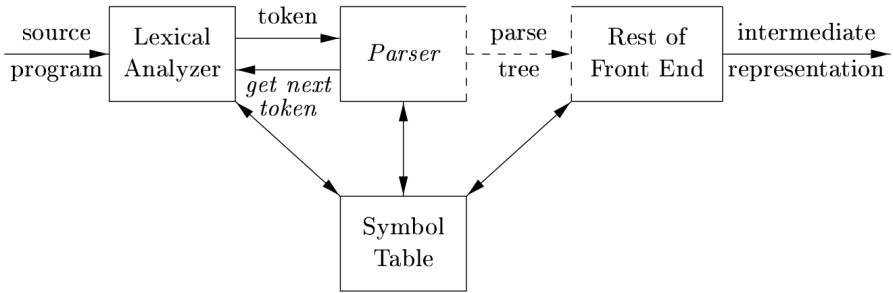
[1] syntax

تحلیل نحوی

- توصیف ساختار نحوی توسط گرامر مستقل از متن دارای مزیت‌های زیر است:
- گرامرها می‌توانند توصیف بسیار دقیق و قابل فهمی از یک زبان برنامه‌نویسی ارائه می‌کنند.
- همچنین ابزارهایی وجود دارند که قادرند با دریافت گرامر یک زبان، به طور خودکار یک تجزیه کننده تولید کنند. استفاده از چنین ابزارهایی کمک می‌کنند که در صورتی که گرامر مشکلاتی داشته باشد، مشکلات آن به طور خودکار تشخیص داده شوند. برای مثال طراح یک گرامر ممکن است قادر به تشخیص ابهام در گرامر نباشد، درحالی که ابزار ممکن است ابهام‌ها را تشخیص دهد.
- یک طراح کامپایلر نیاز دارد برای طراحی درست تجزیه کننده توصیف دقیقی از زبان مورد نظر داشته باشد.
- وقتی یک کامپایلر براساس قوانین یک گرامر ساخته شده باشد، با تغییر گرامر به سادگی می‌توان برنامه تجزیه کننده کامپایلر را نیز تغییر داد.

تحلیل نحوی

- تحلیل‌گر نحوی [1] یا تجزیه کننده [2] (پارسر) توکن‌ها را از تحلیل‌گر لغوی دریافت می‌کند و بررسی می‌کند آیا دنبالهٔ توکن‌های دریافت شده می‌توانند توسط زبان گرامر توصیف شده تولید شوند یا خیر.



[1] syntax analyzer
[2] parser

تحلیل نحوی

- یک تجزیه کننده همچنین معمولاً قادر است هرنوع خطای نحوی را گزارش کرده و ادامهٔ رشتهٔ ورودی را پس از خطا تجزیه کنند.
- یک تجزیه کننده با دریافت توکن‌ها، یک درخت تجزیه تولید می‌کند و درخت تجزیه تولید شده را به قسمت بعدی کامپایلر ارسال می‌کند.

تحلیل نحوی

- سه دسته از تجزیه کننده‌ها برای گرامرها وجود دارند : تجزیه کننده‌های عمومی [1] ، بالا به پایین [2] ، و پایین به بالا [3] .
- الگوریتم‌های تجزیه عمومی مانند الگوریتم سی‌وای‌کا [4] و الگوریتم ایرلی [5] می‌توانند هرنوع الگوریتم مستقل از متن را تجزیه کنند. مشکل اصلی این تجزیه کننده‌ها این است که پیچیدگی زمانی بالایی دارند. گرچه پیچیدگی سی‌وای‌کا $O(n^3)$ و پیچیدگی الگوریتم ایرلی در بدترین حالت $O(n^3)$ است و از لحاظ تئوری پیچیدگی پایینی به حساب می‌آید ولی در عمل برای پیاده‌سازی کامپایلرها به تجزیه کننده‌هایی نیاز داریم که پیچیدگی زمانی پایین‌تری داشته باشند.

[1] universel
[2] top-down
[3] bottom-up
[4] Cocke-Younger-kasami (CYK)
[5] Earley

تحلیل نحوی

- معمولاً در کامپایلرها از تجزیه کننده‌های بالا به پایین و پایین به بالا استفاده می‌شود.
- همانطور که از اسم این تجزیه کننده‌ها مشخص است، تجزیه کننده‌های بالا به پایین درخت تجزیه را از ریشه به برگ می‌سازند، درحالی که تجزیه کننده‌های پایین به بالا از برگ‌های درخت تجزیه آغاز می‌کنند تا به ریشه درخت برسند و درخت تجزیه را تشکیل دهند.
- در هر صورت ورودی تجزیه کننده دنباله‌ای از توکن‌هاست که از چپ به راست خوانده می‌شود.

تحلیل نحوی

- تجزیه کننده‌های بالا به پایین و پایین به بالا برای زیر مجموعه‌ای از گرامرهای مستقل از متن کارایی دارند، اما برخی از این گرامرهای خاص به خصوص گرامرهای LL و LR برای توصیف همهٔ ساختارهای زبان‌های برنامه‌نویسی موجود کافی هستند.

تحلیل نحوی

- معمولاً بسیاری از ساختارهای زبان‌های برنامه‌نویسی پیچیدگی خاصی برای تجزیه ندارند. برای مثال یک حلقه `while` در زبان جاوا از کلمه `while` ، یک عبارت درون یک جفت پرانتز و یک جفت آکولاد تشکیل شده است.
- عبارات ریاضی معمولاً به علت اولویت و وابستگی عملگرها پیچیدگی بیشتری دارند. بنابراین در اینجا برروی عبارات ریاضی تمرکز می‌کنیم.
- با در نظر گرفتن تنها عملگرهای جمع، ضرب، و پرانتز، یک عبارت [1] به نام E تشکیل شده است از مجموع تعدادی جمله [2] به نام T که با عملگر + با یکدیگر جمع شده‌اند و هریک از جملات تشکیل شده است از ضرب تعدادی فاکتور (ضریب) [3] به نام F که با استفاده از عملگر * در یکدیگر ضرب شده‌اند. هریک از فاکتورها می‌تواند یک شناسه باشد، ویا خود یک عبارت باشد که در بین دو پرانتز قرار گرفته است.

[1] expression
[2] term
[3] factor

تحلیل نحوی

- بنابراین می‌توانیم گرامری به صورت زیر برای توصیف یک عبارت بنویسیم.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

- این گرامر به دسته گرامرهای LR تعلق دارد. این نوع گرامرها را معمولاً توسط تجزیه کننده پایین به بالا تجزیه می‌کنیم.

- این گرامر را نمی‌توانیم توسط تجزیه‌کننده بالا به پایین تجزیه کنیم زیرا بازگشتی چپ [1] است. در یک گرامر بازگشتی چپ قانونی وجود دارد که در آن متغیر سمت چپ بدنه قانون با متغیر سمت چپ قانون برابر است. خواهیم دید که تجزیه‌کنندهٔ بالا به پایین نمی‌تواند گرامرهایی که بازگشت چپ دارند را تجزیه کند.

[1] left recursive

تحلیل نحوی

- گرامر زیر معادل گرامر قبل و غیربازگشتی چپ [1] است و می‌توانیم از یک تجزیه کننده بالا به پایین برای تجزیه برنامه‌ها توسط آن استفاده کنیم. در مورد روش حذف بازگشت چپ توضیح خواهیم داد.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

- همچنین گرامر زیر یک گرامر مبهم است که برای رشته $a + b * c$ بیشتر از یک درخت تجزیه می‌سازد. گرامرهای مبهم را نیز قبل از تجزیه باید رفع ابهام کنیم.

$$E \rightarrow E + E \mid E * E \mid (\ E\ ) \mid \mathbf{id}$$

[1] non-left-recursive

مدیریت خطاهای نحوی

- اگر قرار بود کامپایلرها فقط برنامه‌های درست را تجزیه کنند، طراحی و پیاده سازی آنها بسیار ساده‌تر می‌شد. اما، یک کامپایلر باید علاوه بر کامپایل برنامه به برنامه‌نویس کمک کند مکان و نوع خطاهای برنامه خود را شناسایی کند.

مدیریت خطاهای نحوی

- خطاهای برنامه‌نویسی می‌توانند انواع مختلفی داشته باشند.

۱. خطاهای لغوی مانند خطا در نوشتن نام شناسه‌ها، کلمات کلیدی و غیره.

۲. خطاهای نحوی مانند خطا در نوشتن اشتباه ساختار دستورات.

۳. خطاهای معنایی مانند خطا در انتساب مقدار متغیرها با نوع متفاوت.

۴. خطاهای منطقی که شامل خطاهایی می‌شوند که در یک برنامه اتفاق می‌افتند هنگامی که برنامه از نظر لغوی و نحوی و معنایی درست است و برنامه به درستی کامپایل می‌شود اما نتیجه برنامه با مقدار مورد انتظار برنامه‌نویس متفاوت است. برای مثال در زبان سی ممکن است به اشتباه برنامه‌نویس به اشتباه به جای عملگر تساوی از عملگر انتساب استفاده کند.

مدیریت خطاهای نحوی

- وقتی یک پارسر با خطا مواجه شد می‌تواند کامپایل را متوقف کند و اولین خطایی که با آن مواجه شده است را گزارش کند. اما بهتر است کامپایلر همهٔ خطاهای یک برنامه را با یک بار تجزیه کد تشخیص دهد. برای این کار لازم است پس از مواجه شدن با یک خطا، تجزیه کننده خود را بازیابی کند و تجزیه برنامه را ادامه دهد.

- بنابراین کامپایلر باید علاوه بر تشخیص خطا و گزارش خطا به طور دقیق، بتواند سریعاً پس از رخداد یک خطا بازیابی شده و بررسی برنامه را ادامه دهد تا خطاهای بعدی را تشخیص دهد. همچنین مدیریت خطا نباید سربار زیادی بر روند کامپایل داشته باشد و باعث کندی بیش از اندازه کامپایل برنامه‌ها شود.

مدیریت خطاهای نحوی

- چند استراتژی برای بازیابی از خطا وجود دارد که به آنها اشاره می‌کنیم.
- بازیابی با توکن همگام کننده یا بازیابی اضطراری [1] : در این روش تجزیه کننده از توکن‌ها یک‌به‌یک چشم پوشی می‌کند تا به یکی از توکن‌های همگام کننده [2] برسد. برای مثال علامت آکولاد بسته (}) یا نقطه ویرگول (;) می‌توانند توکن‌های همگام کننده باشند. مشکل این روش این است که ممکن است تعداد زیادی از خطاها نادیده گرفته شوند اما مزیت آن سادگی پیاده‌سازی آن است.
- بازیابی با جایگزینی توکن‌ها [3] : با رخداد خطا، تجزیه‌کننده می‌تواند توکن‌های بعدی در ورودی را جایگزین کند تا جایی که ادامه رشته معنی‌دار و قابل تجزیه باشد. برای مثال با تبدیل یک علامت ویرگول به نقطه ویرگول ممکن است ورودی معنی‌دار و قابل تجزیه شود.

[1] panic-mode recovery
[2] synchronizing tokens
[3] phrase-level recovery

مدیریت خطاهای نحوی

- قوانین گرامری تشخیص خطا [1] : با پیش‌بینی کردن خطاهای معمول برنامه‌نویسی می‌توان تعدادی قوانین گرامری به گرامر اضافه کرد که خطاها را تشخیص می‌دهند.
- تصحیح عمومی و بهینه [2] : معمولاً انتظار داریم تجزیه‌کننده کمترین تعداد تصحیح را در ورودی انجام دهد. الگوریتم‌هایی وجود دارند که می‌توانند از بین چندین روش برای تصحیح گرامر، گزینه‌ای را انتخاب کنند که با استفاده از آن کمترین تصحیح بر روی ورودی صورت گیرد. این الگوریتم‌ها معمولاً بسیار پرهزینه هستند و معمولاً در عمل استفاده نمی‌شوند.

[1] error production rules
[2] global correction

گرامرهای مستقل از متن

- گرامرهای مستقل از متن می‌توانند ساختار نحوی زبان‌های برنامه‌نویسی را توصیف کنند. این گرامرها به ازای هریک از مفاهیم در زبان برنامه‌نویسی یک متغیر تعریف می‌کنند.
- برای مثال اگر مفاهیم دستور [1] ($stmt$) و عبارت [2] ($expr$) را در نظر بگیریم، می‌توانیم قانون گرامر زیر را تعریف کنیم.

```
stmt → if (expr) stmt else stmt
```

- با استفاده از قوانین دیگر می‌توانیم تعریف کنیم یک دستور چه شکل‌های دیگری می‌تواند داشته باشد.

[1] statement
[2] expression

گرامرهای مستقل از متن

- یک گرامر مستقل از متن تشکیل شده است از نمادهای پایانی یا ترمینال‌ها، نمادهای غیرپایانی یا متغیرها، یک نماد آغازین و تعدادی قوانین تولید.

۱. ترمینال‌ها [1] یا نمادهای پایانی یا پایانه‌ها واحدهایی هستند که رشته ورودی را تشکیل می‌دهند. ترمینال‌ها در یک گرامر یک زبان برنامه‌نویسی همان توکن‌ها هستند. برای مثال کلمات کلیدی `if` و `else` و کاراکترهای) و (ترمینال‌های یک گرامر هستند.

۲. نمادهای غیرپایانی [2] یا غیرپایانه‌ها یا متغیرها دنباله‌ای از توکن‌ها را با یک نام انتزاعی نامگذاری می‌کنند. برای مثال متغیر $stmt$ نمایندهٔ مفهوم دستور است که مقدار آن می‌تواند هریک از دستورات زبان باشد.

[1] terminal
[2] nonterminal

گرامرهای مستقل از متن

۳. یکی از متغیرها به عنوان نماد آغازین [1] استفاده می‌شود.

۴. قوانین تولید [2] یک گرامر تعیین می‌کنند چگونه متغیرها و ترمینال‌ها در کنار یکدیگر قرار می‌گیرند تا یک رشته از یک زبان را تشکیل دهند. هر قانون تولید تشکیل شده است از یک متغیر سمت چپ یا متغیر قانون تولید یا متغیر ابتدای قانون تولید [3] ، یک نماد $\rightarrow$ که گاهی با $::=$ نشان داده می‌شود و یک بدنه یا سمت راست [4] قانون که از صفر یا چند ترمینال و متغیر تشکیل شده است. در فرایند تجزیهٔ یک رشته با متغیر آغازین شروع می‌کنیم و متغیر را با بدنه یکی از قوانین تولید مربوط به آن جایگزین می‌کنیم. این فرایند را ادامه می‌دهیم تا رشته به دست بیاید. در صورتی که رشته مورد نظر به دست نیامد، رشته عضو گرامر آن زبان نیست. مجموعهٔ همهٔ رشته‌هایی که با شروع از نماد آغازین و اعمال قوانین یک گرامر به دست می‌آیند، زبان آن گرامر را تعیین می‌کنند.

---

[1] start symbol
[2] production rule
[3] head or left side
[4] body or right side

گرامرهای مستقل از متن

- برای مثال گرامر زیر عبارات ریاضی را تجزیه می‌کند که شامل عملگرهای + و - و * و / و (و) هستند. کلمه `id` درواقع نوع توکن شناسه است که در تحلیل لغوی استخراج شده است.

$$
\begin{aligned}
expression &\rightarrow expression \texttt{ + } term \\
expression &\rightarrow expression \texttt{ - } term \\
expression &\rightarrow term \\
term &\rightarrow term \texttt{ * } factor \\
term &\rightarrow term \texttt{ / } factor \\
term &\rightarrow factor \\
factor &\rightarrow \texttt{(} expression \texttt{)} \\
factor &\rightarrow \mathbf{id}
\end{aligned}
$$

گرامرهای مستقل از متن

- در این فصل از علائم و نشانه‌گذاری‌های زیر استفاده می‌کنیم.
- ترمینال‌ها شامل موارد زیر هستند : حروف کوچک ابتدایی الفبای انگلیسی مانند a و b و c، عملگرها مانند $+$ و $*$ ، علائم نشانه‌گذاری مانند پرانتز و کاما ، ارقام مانند 0 و 1 و $\cdots$ و 9 و رشته‌های پررنگ مانند **id** و **if** .
- متغیرها شامل موارد زیر هستند : حروف بزرگ ابتدایی الفبای انگلیسی مانند A و B و C ، حرف S که بیشتر به عنوان متغیر آغازین استفاده می‌شود، رشته‌هایی که به صورت مورب نوشته می‌شود مانند $expr$ و $stmt$.

گرامرهای مستقل از متن

- معمولاً وقتی می‌خواهیم از یک نماد گرامر، که ممکن است ترمینال یا متغیر باشد، صحبت کنیم آن را با حروف X و Y و Z نمایش می‌دهیم.
- یک رشته شامل ترمینال‌ها را معمولاً با حروف u و v و w و z نمایش می‌دهیم.
- برای نمایش دنباله‌ای از ترمینال‌ها و متغیرها از حروف یونانی مانند α و β استفاده می‌کنیم. مثلاً $A \rightarrow \alpha$ یک قانون گرامر است.
- وقتی یک متغیر چندین بدنه داشته باشد آنها را با علامت خط عمودی از یکدیگر جدا می‌کنیم مثلاً $A \rightarrow \alpha_1|\alpha_2|\cdots|\alpha_k$
- معمولاً متغیر سمت چپ اولین قانون همان متغیر آغازین است.

گرامرهای مستقل از متن

- گرامر زیر عبارات ریاضی را توصیف می‌کند که در آن از متغیرهای E و T و F و ترمینال‌های +، -، *، /،)، (، و `id` استفاده شده است.

$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

اشتقاق

- به فرایندی که در آن یک رشته توسط قوانین یک گرامر تولید می‌شود، فرایند اشتقاق [1] گفته می‌شود.
- با شروع از نماد آغازین، در هرگام یکی از متغیرها با بدنه یکی از قوانین متعلق به آن متغیر جایگزین می‌شود. دنباله ترمینال‌ها و متغیرهایی که در هرگام به دست می‌آید را یک صورت جمله‌ای [2] می‌نامیم. اگر با جایگزین کردن متغیرها در صورت‌های جمله‌ای توسط بدنه قوانین متعلق به آنها، رشته مورد نظر به دست آمد، آن رشته متعلق به زبان گرامر است. در این صورت می‌گوییم رشته توسط گرامر مشتق می‌شود یا تولید می‌شود یا به دست می‌آید.

[1] derivation
[2] sentential form

اشتقاق

- برای مثال، گرامر زیر با یک متغیر E را در نظر بگیرید.

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$

- فرض کنید می‌خواهیم جمله $-(\mathrm{id})$ توسط این گرامر به دست آوریم. می‌توانیم با اعمال سه قانون این رشته را به دست آوریم. می‌گوییم E با استفاده از قانون سوم مشتق می‌کند یا به دست می‌دهد $-E$ و سپس با استفاده از قانون چهارم به دست می‌دهد $-(E)$ و در نهایت با استفاده از قانون پنجم به دست می‌دهد $-(\mathrm{id})$.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathrm{id})$$

- به این دنباله از جایگزینی متغیرها یا بدنهٔ قوانین متعلق به آنها یک فرایند اشتقاق می‌گوییم.

اشتقاق

- دنباله‌ای از نمادها به صورت $\alpha A \beta$ را در نظر بگیرید به طوری که α و β دنباله‌ای از نمادهای پایانی و غیرپایانی (ترمینال‌ها و متغیرهای) گرامر هستند و A یک نماد غیرپایانی (متغیر) است.
- فرض کنید $A \rightarrow \gamma$ یک قانون تولید باشد. آنگاه می‌نویسیم $\alpha A \beta \Rightarrow \alpha \gamma \beta$. نماد $\Rightarrow$ به معنی مشتق کردن در یک گام [1] است.

[1] derives in one step

اشتقاق

- وقتی دنباله‌ای به صورت $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \alpha_n$ داشته باشیم به طوری که $n \geq 1$، می‌گوییم α_n از α_1 مشتق می‌شود و یا α_1 به دست می‌دهد α_n و یا α_1 در صفر یا چند گام مشتق می‌کند α_n. به عبارت دیگر :

۱. به ازای هر صورت جمله‌ای α داریم $\alpha \overset{*}{\Rightarrow} \alpha$ یعنی هر صورت جمله‌ای می‌تواند خود را در صفر یا چندگام [1] مشتق کند.

۲. اگر $\alpha \overset{*}{\Rightarrow} \beta$ و $\beta \Rightarrow \gamma$ آنگاه $\alpha \overset{*}{\Rightarrow} \gamma$

- همچنین گاهی می‌نویسیم $\overset{+}{\Rightarrow}$ به معنی مشتق کردن در یک یا چند گام.

- اگر $S \overset{*}{\Rightarrow} \alpha$ جایی که S نماد آغازین گرامر G است، می‌گوییم α یک صورت جمله‌ای [2] از گرامر G است.

- یک صورت جمله‌ای شامل متغیرها و ترمینال‌هاست. یک جمله [3] از یک گرامر یک صورت جمله‌ای است که در آن هیچ متغیری نباشد.

[1] derives in zero or more steps
[2] sentential form
[3] sentence

اشتقاق

- زبان تولید شده توسط یک گرامر مجموعه‌ای است از همهٔ جمله‌های تولید شده توسط آن گرامر.
- رشته w در زبان تولید شده توسط گرامر G یا $L(G)$ است اگر و تنها اگر w یک جمله از گرامر G باشد یا به عبارت دیگر $S \stackrel{*}{\Rightarrow} w$.
- زبانی که توسط یک گرامر مستقل از متن تولید می‌شود، یک زبان مستقل از متن نام دارد.
- اگر دو گرامر، یک زبان یکسان تولید کنند، آن دو گرامر معادل یکدیگرند.

اشتقاق

- گرامر زیر را در نظر بگیرید.

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$

- جمله $-(\mathrm{id} + \mathrm{id})$ یک جمله از این گرامر است زیرا فرایند اشتقاق زیر برای آن وجود دارد :

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathrm{id} + E) \Rightarrow -(\mathrm{id} + \mathrm{id})$$

- می‌نویسیم $E \stackrel{*}{\Rightarrow} -(\mathrm{id} + \mathrm{id})$ و می‌خوانیم جمله $-(\mathrm{id} + \mathrm{id})$ از متغیر E مشتق می‌شود.
- در هر گام در فرایند اشتقاق دو انتخاب وجود دارد. باید انتخاب کنیم کدام متغیر را جایگزین کنیم و همچنین کدام قانون متعلق به متغیر انتخاب شده را انتخاب کنیم.

اشتقاق

- دو نوع فرایند اشتقاق را به صورت زیر تعریف می‌کنیم :

۱. در اشتقاق چپ [1] ، متغیری که در صورت جمله‌ای در سمت چپ بقیه متغیرها قرار دارد و به عبارت دیگر چپ‌ترین [2] است، انتخاب می‌شود. اگر $\alpha \Rightarrow \beta$ گامی باشد که در آن چپ‌ترین متغیر α انتخاب شود، می‌نویسیم $\alpha \underset{lm}{\Rightarrow} \beta$.

۲. در اشتقاق راست [3] ، متغیری که راست‌ترین [4] است انتخاب می‌شود و می‌نویسیم $\alpha \underset{rm}{\Rightarrow} \beta$.

[1] leftmost derivation
[2] leftmost
[3] rightmost derivation
[4] rightmost

اشتقاق

- برای مثال :

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$$

- به طور خلاصه می‌گوییم $w A \gamma \underset{lm}{\Rightarrow} w \delta \gamma$ جایی که w فقط از ترمینال‌ها تشکیل شده و $A \to \delta$ یک قانون تولید است و γ رشته‌ای است تشکیل شده از متغیرها و ترمینال‌ها.
- اگر $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$ آنگاه می‌گوییم α یک صورت جمله‌ای چپ [1] از گرامر است.

[1] left sentential form

درخت تجزیه

- درخت تجزیه [1] یک نمایش گرافیکی از فرایند تجزیه است که در آن ترتیب جایگزینی متغیرها نشان داده نمی‌شود.
- هر رأس میانی در درخت تجزیه، اِعمال یک قانون در فرایند اشتقاق را نشان می‌دهد. اگر یک رأس با برچسب A در درخت تجزیه داشته باشیم، فرزندان آن از سمت چپ به راست به ترتیب ترمینال‌ها و متغیرهایی هستند که در بدنه یکی از قوانین متعلق به A قرار دارند.

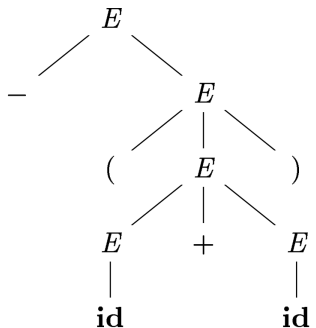
[1] parse tree

درخت تجزیه

- گرامر زیر را در نظر بگیرید.

$$\mathrm{E} \rightarrow \mathrm{E}+\mathrm{E} \mid \mathrm{E} * \mathrm{E} \mid -\mathrm{E} \mid (\mathrm{E}) \mid \mathbf{id}$$

- درخت تجزیه زیر برای به دست آوردن رشته $-(\mathrm{id}+\mathrm{id})$ با استفاده از این گرامر تشکیل شده است.



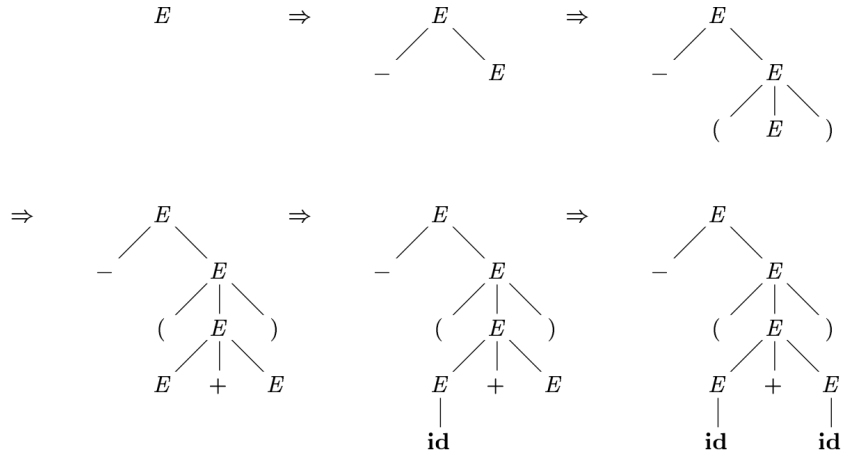
درخت تجزیه

- برگ‌های درخت تجزیه همه با ترمینال‌ها برچسب زده شده‌اند و به ترتیب از چپ به راست رشته‌ای را تشکیل می‌دهند که توسط گرامر مشتق شده است.
- به رشته‌ای که از الحاق برگ‌های درخت تجزیه از چپ به راست به دست می‌آید محصول [1] درخت تجزیه گفته می‌شود.
- یک درخت تجزیه می‌تواند تجزیه یک صورت جمله‌ای را نشان دهد. به درخت تجزیه‌ای که محصول آن یک صورت جمله‌ای باشد، درخت تجزیه جزئی [2] نیز گفته می‌شود. به درخت تجزیه‌ای که محصول آن یک جمله باشد، درخت تجزیه کامل [3] گفته می‌شود.

[1] yield
[2] partial parse tree
[3] complete parse tree

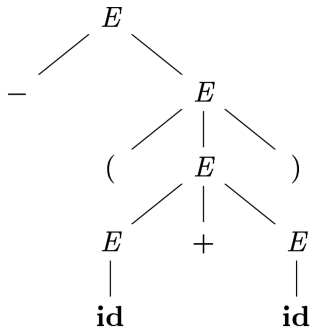
درخت تجزیه

- در شکل زیر درخت‌های تجزیه برای رشتهٔ $-(\mathrm{id}+\mathrm{id})$ در فرایند اشتقاق چپ نشان داده شده‌اند.



درخت تجزیه

- یک درخت تجزیه می‌تواند متناظر با چند فرایند اشتقاق باشد. مثلاً دو فرایند اشتقاق چپ و اشتقاق راست می‌توانند یک درخت تجزیه واحد تولید کنند.
- درخت تجزیهٔ زیر می‌تواند توسط یک اشتقاق چپ یا یک اشتقاق راست تولید شده باشد.



ابهام

- گرامری که بیش از یک درخت تجزیه برای یک جمله تولید کند، مبهم [1] نامیده می‌شود.
- به عبارت دیگر یک گرامر مبهم برای تولید یک رشته بیش از یک فرایند اشتقاق چپ (یا بیش از یک فرایند اشتقاق راست) دارد.

[1] ambiguous

ابهام

- گرامر زیر را در نظر بگیرید.

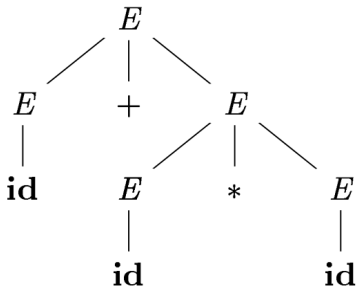
$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

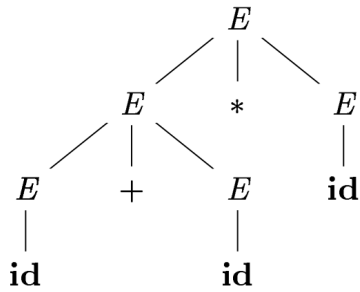
- برای به دست آوردن رشته $\mathtt{id} + \mathtt{id} * \mathtt{id}$ توسط این گرامر دو فرایند اشتقاق چپ وجود دارد.

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

ابهام

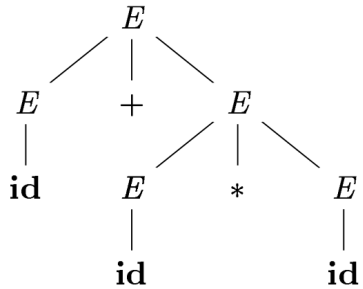
- همچنین برای این رشته دو درخت تجزیه به صورت زیر وجود دارد.

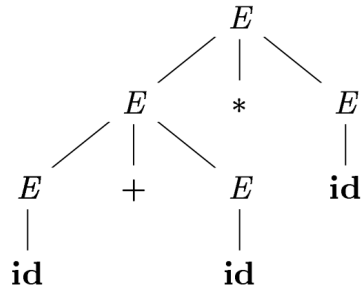




ابهام

- دقت کنید که این دو درخت تجزیه دو معنی متفاوت از رشته تولید شده به دست می‌دهند. اگر بخواهیم رشته $a + b * c$ را توسط این گرامر تجزیه کنیم، درخت سمت چپ معادل $a + (b * c)$ و درخت سمت راست معادل $(a + b) * c$ خواهد بود.





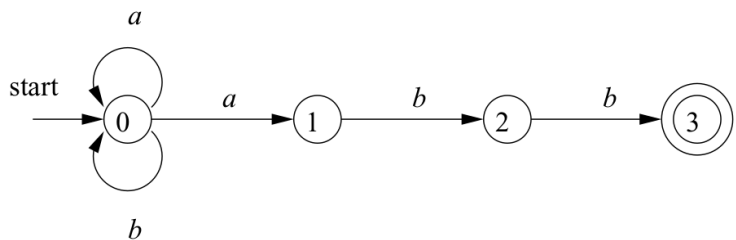
گرامرهای منظم

- گرامرها ابزار قوی‌تری نسبت به عبارات منظم هستند. هر عبارت منظم را می‌توان توسط یک گرامر نشان داد ولی هر گرامر را نمی‌توان توسط یک عبارت منظم نمایش داد. دسته‌ای از گرامرها که برای توصیف زبان‌های منظم به کار می‌روند، گرامرهای منظم نامیده می‌شوند. گرامرهای منظم زیر مجموعه‌ای از گرامرهای مستقل از متن هستند.
- عبارت منظم $(\mathtt{a}|\mathtt{b})^*\mathtt{abb}$ را می‌توان توسط گرامر منظم زیر توصیف کرد.

$$
\begin{aligned}
A_0 &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\
A_1 &\rightarrow bA_2 \\
A_2 &\rightarrow bA_3 \\
A_3 &\rightarrow \epsilon
\end{aligned}
$$

گرامرهای منظم

- الگوریتمی وجود دارد که توسط آن می‌توان یک ماشین متناهی غیرقطعی را به یک گرامر تبدیل کرد.
- گرامر قبل درواقع از ماشین متناهی غیرقطعی زیر به دست می‌آید.



گرامرهای منظم

- این الگوریتم به صورت زیر عمل می‌کند :

۱. به ازای هر حالت i از ماشین متناهی غیرقطعی متغیر A_i را می‌سازیم.

۲. اگر حالت i با ورودی a به حالت j می‌رود آنگاه قانون $A_i \to aA_j$ را به گرامر اضافه می‌کنیم. اگر حالت i با ورودی ϵ به حالت j می‌رود آنگاه قانون $A_i \to A_j$ را به گرامر اضافه می‌کنیم.

۳. اگر حالت i یک حالت نهایی است آنگاه قانون $A_i \to \epsilon$ را اضافه می‌کنیم.

۴. اگر حالت i یک حالت شروع است، آنگاه A_i را متغیر آغازین قرار می‌دهیم.

گرامرهای منظم

- برخی از زبان‌ها را نمی‌توانیم توسط یک گرامر منظم توصیف کنیم. این زبان‌ها متعلق به دستهٔ زبان‌های منظم نیستند و ماشین متناهی برای آنها وجود ندارد.
- برای مثال $L = \{a^n b^n | n \geqslant 1\}$ زبانی است که نمی‌توان برای توصیف آن از یک ماشین متناهی استفاده کرد. توسط لم تزریق اثبات می‌شود که این زبان متعلق به دسته زبان‌های منظم نیست، اما می‌توان آن را توسط یک گرامر مستقل از متن توصیف کرد.

گرامرهای منظم

- گرامرهای مستقل از متن می‌توانند زبان‌های برنامه‌نویسی را توصیف کنند. البته گرامرها قادر به توصیف معنایی زبان‌ها نیستند. برای مثال توسط گرامر مستقل از متن نمی‌توانیم نیاز یک متغیر به تعریف قبل از استفاده از آن را توصیف کنیم.
- برای این که یک گرامر برای تجزیه کننده قابل استفاده باشد، باید پردازش‌هایی برروی آن انجام شود که در اینجا به آنها اشاره می‌کنیم. برای مثال یک گرامر ابتدا باید رفع ابهام شود. سپس برای استفاده در تجزیه کننده بالا به پایین باید بازگشت چپ در آن حذف شود.

گرامرهای منظم

- همانطور که گفته شد، زبان‌های منظم را نیز می‌توان توسط گرامرها توصیف کرد. سؤالی که در اینجا ممکن است به وجود آید این است که چرا نیاز است که یک تحلیل‌گر لغوی قبل از تحلیل‌گر نحوی داشته باشیم؟

- با جدا کردن تحلیل‌گر لغوی از تحلیل‌گر نحوی تجزیه کننده بسیار ساده‌تر می‌شود و برنامه کامپایلر ساده‌تر می‌شود که باعث می‌شود تعداد خطاهای برنامه‌نویسی در نوشتن کامپایلر کاهش پیدا کند و همچنین برنامه کامپایلر ساده‌تر شود و راحت‌تر بتوان آن را تغییر داد. همچنین قوانین در تحلیل‌گر لغوی نسبتا ساده‌اند و با عبارت‌های منظم ساده تولید می‌شوند و نیازی به افزودن گرامرهای پیچیده برای آنها وجود ندارد. به علاوه روشی وجود دارد که تحلیل‌گر لغوی مستقیما از عبارت منظم تولید می‌شود. به این دلایل تحلیل‌گر لغوی از تحلیل‌گر نحوی جدا می‌شود.

رفع ابهام

- گاهی می‌توانیم یک گرامر غیرمبهم معادل یک گرامر مبهم بنویسیم. اما این کار همیشه ممکن نیست زیرا برخی از زبان‌ها ذاتا مبهم هستند.
- گرامر مبهم زیر را در نظر بگیرید.

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

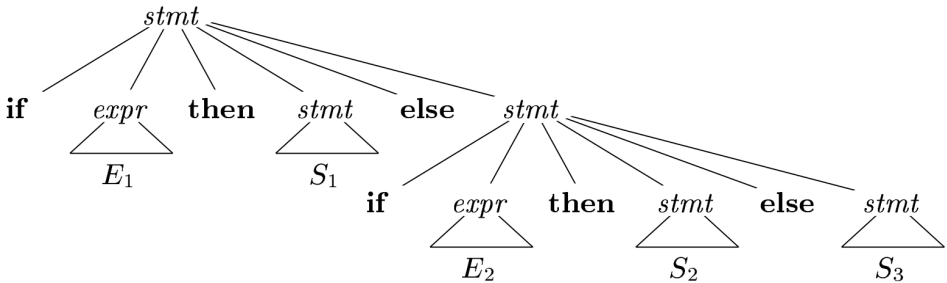
- در اینجا `other` به معنی هر دستور دیگری به غیر از دستورات شرطی `if-else` است.

رفع ابهام

- با استفاده از این گرامر می‌توانیم جمله زیر را تولید کنیم.

$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$

- برای این جمله درخت تجزیه زیر وجود دارد.



رفع ابهام

- حال جمله زیر را در نظر بگیرید.

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

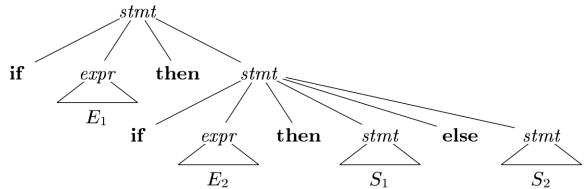
- درخت تجزیه‌ای برای این جمله رسم کنید.

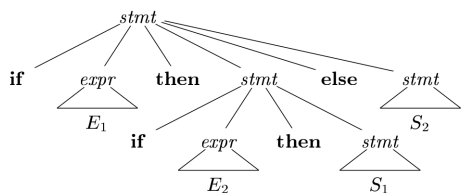
رفع ابهام

- حال جمله زیر را در نظر بگیرید.

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

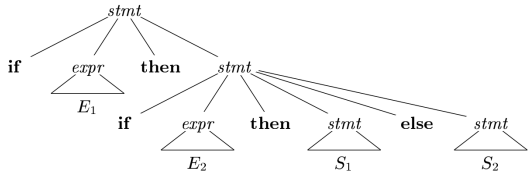
- برای این جمله دو درخت تجزیه به صورت زیر وجود دارد.

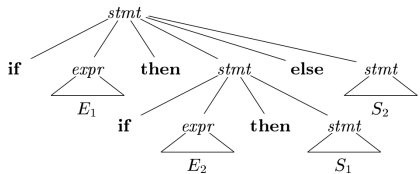




رفع ابهام

- در همهٔ زبان‌های برنامه‌نویسی درخت تجزیه اول را به درخت دوم ترجیح می‌دهیم.





- در واقع قانونی که در همه زبان‌ها وجود دارد این است که `else` به نزدیک‌ترین `if` (یا `then`) قبل از آن تعلق دارد.

رفع ابهام

- می‌توانیم یک گرامر غیرمبهم به صورت زیر تولید کنیم که معادل گرامر مبهم ذکر شده است.
- توجه کنید که بین `then` و `else` اگر قرار باشد دستور شرطی `if` قرار بگیرد، باید حتماً یک `if-then-else` باشد، در غیراینصورت ابهام به وجود می‌آید.
- در واقع قانونی که برای گرامر غیرمبهم وضع می‌کنیم این است که همیشه بین `then` و `else` یا یک عبارت `if-then-else` قرار می‌گیرد و یا یک دستور غیرشرطی. اما بعد از `else` ممکن است یک عبارت شرطی بدون `else` به کار رفته شود.

رفع ابهام

- گرامر زیر معادل گرامر شرطی برای دستورات `if-then-else` است که ابهام در آن رفع شده است.

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{matched\_stmt} \\
& | & \mathit{open\_stmt} \\
\mathit{matched\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{matched\_stmt} \\
& | & \textbf{other} \\
\mathit{open\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
& | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{open\_stmt}
\end{array}
$$

تجزیه بالا به پایین

- قبل از بررسی مفصل تجزیه‌کنندهٔ بالابه‌پایین، برای یک گرامر ساده که زیر مجموعه‌ای از گرامر زبان جاوا و سی است، یک تجزیه کنندهٔ بالا به پایین [1] می‌سازیم و سپس در مورد روند کلی ساختن تجزیه کنندهٔ بالا به پایین صحبت می‌کنیم.
- گرامر زیر را در نظر بگیرید.

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textbf{expr ;} \\
 & | & \textbf{if (expr)} \ \mathit{stmt} \\
 & | & \textbf{for (} \ \mathit{optexpr} \ \textbf{;} \ \mathit{optexpr} \ \textbf{;} \ \mathit{optexpr} \ \textbf{)} \ \mathit{stmt} \\
 & | & \textbf{other} \\
 & & \\
\mathit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

- در اینجا **expr** و **other** را به عنوان دو ترمینال در نظر گرفتیم. در یک گرامر کامل این دو را به عنوان دو متغیر در نظر می‌گیریم و توسط قوانین دیگر تعریف می‌کنیم.

[1] top-down parser

تجزیه بالا به پایین

- تجزیه کنندهٔ بالا به پایین یک درخت تجزیه با یک ریشه می‌سازد به طوری که برچسب ریشهٔ درخت متغیر آغازین گرامر ($stmt$) است.
- تجزیه کنندهٔ بالا به پایین به طور خلاصه به طور مکرر عملیات زیر را انجام می‌دهد.

۱. در رأس N با برچسب A، یکی از قوانین تولید متغیر A را انتخاب می‌کند و فرزندان N را نمادهای (متغیرها و ترمینال‌های) بدنهٔ قانون انتخاب شده قرار می‌دهد.

۲. رأس بعدی در درخت تجزیه که با یک متغیر بر چسب زده است و چپ‌ترین متغیر در بین همهٔ برگ‌هاست را انتخاب می‌کند و آن برگ را با توجه به رشتهٔ ورودی گسترش می‌دهد.

- در هرگام از فرایند تجزیه، تجزیه کننده با توجه به توکن بعدی [1] در رشته ورودی تصمیم می‌گیرد چه قانونی را انتخاب کند.

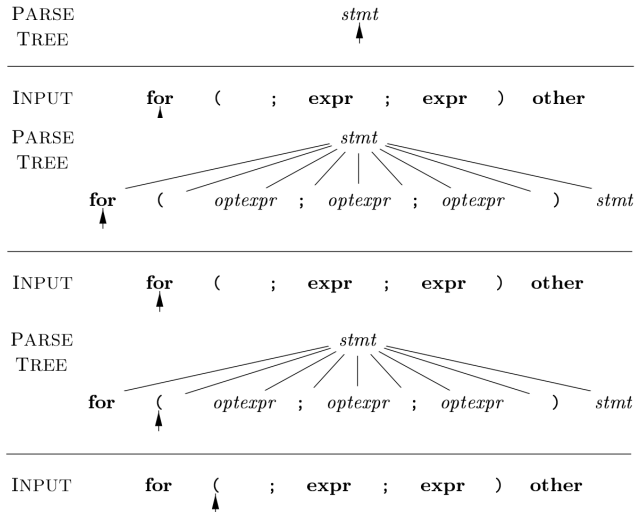
[1] lookahead

تجزیه بالا به پایین

- در تجزیهٔ رشته `for (;expr;expr) other` اولین توکن، واژه `for` است. بنابراین ریشهٔ درخت تجزیه که با *stmt* برچسب زده شده است با قانونی از متغیر *stmt* گسترش می‌یابد که بدنهٔ آن با واژهٔ `for` آغاز شده است.
- در گام بعد در درخت تجزیه باید برگی را تجزیه کنیم که بعد از برگ با برچسب `for` قرار دارد. این برگ) است و در رشته ورودی نیز توکن بعدی نماد) است. در اینجا نماد درخت تجزیه بر نماد رشته ورودی منطبق می‌شود و در درخت تجزیه و رشته ورودی باید به سمت نماد بعدی حرکت کنیم.
- در گام بعد نماد بعدی در درخت تجزیه را انتخاب می‌کنیم که اولین رخداد متغیر *optexpr* است. این روند ادامه می‌یابد تا کل رشته ورودی تجزیه شود.

تجزیه بالا به پایین

- شکل زیر روند تجزیه یک رشته توسط تجزیه کننده بالا به پایین را نشان می‌دهد.



تجزیه بالا به پایین

- در حالت کلی در یک تجزیه کننده بالا به پایین ممکن است انتخاب یک قانون با خطا روبرو شود که در این صورت باید با استفاده از پسگرد یا عقبگرد [1] قانون بعدی انتخاب شود.
- معمولا چنین عقبگردهایی پرهزینه است و به دنبال روش‌های تجزیه‌ای هستیم که از چنین عقبگرهایی جلوگیری کنند.

[1] backtrack

تجزیه پیش‌بینی کننده

- تجزیه کاهشی بازگشتی [1] روشی بالا به پایین برای تحلیل نحوی است که در آن مجموعه‌ای از توابع بازگشتی برای پردازش رشته ورودی استفاده می‌شوند. در این تجزیه‌کننده، به ازای هریک از متغیرهای گرامر یک تابع در نظر گرفته می‌شود.
- یکی از انواع ساده تجزیه کاهشی بازگشتی، تجزیه پیش بینی کننده [2] است. در تجزیه پیش بینی کننده از پسگرد جلوگیری می‌شود.

[1] recursive-descent parsing
[2] predictive parsing

تجزیه پیش‌بینی کننده

- یک تجزیه کنندهٔ پیش بینی کننده برای گرامر قبل در زیر نشان داده شده است.

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

تجزیه پیش‌بینی کننده

- در این تجزیه کننده به ازای هر متغیر گرامر یک تابع تعریف می‌شود. بسته به این که توکن بعدی در رشته ورودی چه مقداری دارد، تجزیه کننده، ورودی را با گرامر تطبیق می‌دهد.
- برای تطبیق [1] یک ترمینال در گرامر و یک کلمه از رشته ورودی، تجزیه کننده صرفا بررسی می‌کند که ترمینال گرامر و توکن بعدی در رشته ورودی برابر باشند.
- برای تطبیق یک متغیر در گرامر و یک کلمه از رشته ورودی، تجزیه کننده تابع متناظر با متغیر را فراخوانی می‌کند.

[1] match

تجزیه پیش‌بینی کننده

- تجزیه کنندهٔ پیش بینی کننده برای یک گرامر ساده [1] می‌تواند مورد استفاده قرار بگیرد.
- برای تعریف گرامر ساده، تابع $\text{First}(\alpha)$ را به صورت زیر تعریف می‌کنیم، جایی که α دنباله‌ای از ترمینال‌ها و متغیرهاست.
- اگر اولین کلمه در دنبالهٔ α ترمینال t باشد، آنگاه $\text{First}(\alpha) = \{t\}$.
- اگر اولین کلمه در دنبالهٔ α متغیر A باشد و داشته باشیم $A \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$ آنگاه $\text{First}(\alpha) = \text{First}(\beta_1) \cup \text{First}(\beta_2) \cup \cdots \cup \text{First}(\beta_n)$

[1] simple grammar

تجزیه پیش‌بینی کننده

- حال فرض کنید در گرامر G داشته باشیم $A \to \alpha$ و $A \to \beta$. گرامر G ساده است اگر $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$.
- از تجزیه کننده پیش‌بینی کننده تنها زمانی می‌توان استفاده کرد که یک گرامر ساده باشد. در اینصورت در زمان خطی یک رشته را می‌توان تجزیه کرد.

تجزیه پیش‌بینی کننده

- در تجزیه کننده پیش بینی کننده‌ای که طراحی کردیم، در پیاده‌سازی تابع `optexpr()` در صورتی که تطبیق رخ ندهد خطایی صادر نکردیم. با این کار در واقع قانون $\texttt{optexpr} \to \epsilon$ را پیاده‌سازی کردیم.
- در حالت کلی اگر قانونی به صورت $A \to X_1 \mid X_2 \mid \cdots \mid \epsilon$ داشته باشیم، در پیاده‌سازی تابع `A()` هیچ خطایی صادر نمی‌کنیم، زیرا ممکن است رشته ورودی، در بدنهٔ هیچ یک از قوانین A منطبق نشود که در اینصورت قانون تهی اعمال می‌شود.

تجزیه پیش‌بینی کننده

- برای پیاده‌سازی یک تجزیه کننده پیش بینی کننده برای یک گرامر ساده، به ازای هر یک از متغیرهای گرامر یک تابع تعریف می‌کنیم. با شروع از تابع متعلق به متغیر آغازین تجزیه کننده مکررا به ازای هر متغیر A در گرامر که دارای قوانین $A \to X_1 \mid X_2 \mid \cdots \mid X_n$ است، قانون $A \to X_i$ را انتخاب می‌کند، اگر توکن بعدی در رشته ورودی در مجموعهٔ $\mathrm{First}(X_i)$ باشد.
- با فرض براینکه X_i دنباله‌ای از ترمینال‌های t و متغیرهای X است، به ازای هر ترمینال t، توکن بعدی در رشتهٔ ورودی باید برابر با ترمینال t باشد و به ازای هر متغیر X ، تابع `X()` فراخوانی می‌شود.
- اگر رشته ورودی بدون خطا پایان رسید، رشته متعلق به زبان آن گرامر است.

تجزیه پیش‌بینی کننده

- ممکن است یک تجزیه کننده کاهشی بازگشتی [1] در یک حلقه بی‌پایان بیافتد.
- فرض کنید یک قانون بازگشتی چپ [2] به صورت زیر داشته باشیم :

```
expr → expr + term
```

[1] recursive-descent parser
[2] left-recursive rule

تجزیه پیش‌بینی کننده

- در این قانون، متغیر سمت چپ قانون برابر با نماد سمت چپ در بدنهٔ قانون است.
- حال در فرایند تجزیه اگر تابع `expr()` فراخوانی شود، این تابع نیز مجدداً تابع `expr()` را فراخوانی می‌کند و این فراخوانی بازگشتی خاتمه پیدا نمی‌کند.
- برای استفاده از تجزیه کننده کاهشی بازگشتی باید قوانین بازگشتی چپ را حذف کنیم.

حذف بازگشت چپ

- یک گرامر بازگشتی چپ [1] است اگر به ازای متغیر A و صورت جمله‌ای دلخواه α فرایند اشتقاق $A \overset{+}{\Rightarrow} A\alpha$ وجود داشته باشد.
- تجزیه کننده‌های بالا به پایین نمی‌توانند گرامرهایی که دارای بازگشت چپ هستند را تجزیه کنند، بنابراین بازگشت چپ [2] باید در گرامر حذف شود.

[1] left recursive
[2] left recursion

حذف بازگشت چپ

- گرامر زیر بازگشتی چپ است.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

- پس از حذف بازگشت چپ در این گرامر، گرامر زیر به دست می‌آید.

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \mathbf{id} \end{array}$$

- قوانین تولید $\mathrm{E} \rightarrow \mathrm{E} + \mathrm{T} \mid \mathrm{T}$ با قوانین $\mathrm{E} \rightarrow \mathrm{T}\ \mathrm{E}'$ و $\mathrm{E}' \rightarrow +\mathrm{T}\ \mathrm{E}' \mid \epsilon$ جایگزین می‌شوند.

حذف بازگشت چپ

- بازگشت چپ بلاواسطه [1] می‌تواند توسط روش زیر حذف شود.
- ابتدا قوانین را به صورت زیر مرتب می‌کنیم.

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

به طوری که β_i ها با A آغاز نمی‌شوند.

- سپس قوانین تولید متغیر A را با قوانین زیر جایگزین می‌کنیم.

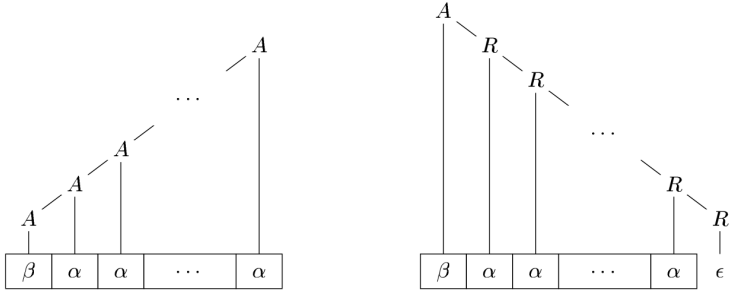
$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

- بدین صورت متغیر A همان رشته‌های قبلی را تولید می‌کند با این تفاوت که بازگشت چپ حذف شده است.

[1] immediate left recursion

حذف بازگشت چپ

- یک قانون بازگشتی چپ به صورت $A \to A\alpha \mid \beta$ درواقع رشته‌هایی به صورت $\beta\alpha^*$ تولید می‌کند. چنین رشته‌هایی را می‌توانیم با گرامر غیر بازگشتی $A \to \beta R$ و $R \to \alpha R \mid \epsilon$ نیز تولید کنیم.



- گرامر جایگزین یک گرامر بازگشتی راست [1] است زیرا قانون $R \to \alpha R$ متغیر R را در سمت راست بدنهٔ قانون دارد. قوانین بازگشتی راست در تجزیهٔ کننده‌های بالا به پایین مشکلی به وجود نمی‌آورند.

[1] right recursive

حذف بازگشت چپ

- روشی که مطرح کردیم بازگشت چپ بلاواسطه را حذف می‌کند اما همه بازگشت‌های چپ را حذف نمی‌کند. برخی مواقع پس از چندگام در فرایند اشتقاق بازگشت چپ به وجود می‌آید.
- برای مثال گرامر زیر را در نظر بگیرید :

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- متغیر S بازگشت چپ بلاواسطه ندارد ولی در فرایند اشتقاق خواهیم داشت $S \Rightarrow Aa \Rightarrow Sda$ که یک بازگشت چپ است.
- الگوریتم زیر برای گرامرهایی که در آنها دور وجود ندارد یعنی اشتقاق $A \overset{+}{\Rightarrow} A$ اتفاق نمی‌افتد و همچنین در آنها قانون تولید تهی یعنی $A \rightarrow \epsilon$ وجود ندارد، بازگشت چپ را حذف می‌کند.

حذف بازگشت چپ

- الگوریتم حذف بازگشت چپ، گرامر G بدون دور و بدون قانون تولید تهی را دریافت می‌کند و یک گرامر معادل بدون بازگشت چپ تولید می‌کند.

1) arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2) **for** (each i from 1 to n) {
3) **for** (each j from 1 to $i-1$) {
4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
5) }
6) eliminate the immediate left recursion among the A_i-productions
7) }

حذف بازگشت چپ

- این الگوریتم به صورت زیر عمل می‌کند.
- به ازای $i = 1$ دو حالت وجود دارد. یا $A_1 \to A_1\alpha$ و یا $A_1 \to A_m\alpha$ به طوری که $m > 1$. در حالت اول در خط ۶ الگوریتم بازگشت چپ بلاواسطه حذف می‌شود. در حالت دوم قانون به همان صورت باقی خواهد ماند.
- به ازای $i = 2$ سه حالت وجود دارد. یا $A_2 \to A_2\alpha$ یا $A_2 \to A_1\alpha$ و یا $A_2 \to A_m\alpha$ به ازای $m > 2$. در حالت اول بازگشت چپ بلاواسطه حذف می‌شود. در حالت دوم قانون به صورت $A_2 \to A_p\alpha$ به ازای $p > 1$ بازنویسی می‌شود و در صورتی که $p = 2$ باشد، بازگشت چپ بلاواسطه حذف می‌شود. در حالت سوم قانون به همان صورت باقی می‌ماند. پس در پایان در هر صورت خواهیم داشت $A_2 \to A_m\alpha$ به ازای $m > 2$.
- در حالت کلی پس از اتمام تکرار i ام در الگوریتم، به ازای همهٔ قوانین A_i خواهیم داشت $A_i \to A_m\alpha$ به طوری که $m > i$.
- در تکرار آخر الگوریتم، بازگشت چپ برای A_n در صورت وجود حذف می‌شود.

حذف بازگشت چپ

- برای استفاده از این الگوریتم ابتدا همه قوانین تولید تهی به صورت $A \to \epsilon$ را حذف می‌کنیم. اگر قانون $A \to \epsilon$ وجود داشته باشد، قانون $S \to AS\alpha$ نیز دارای بازگشت چپ است که این الگوریتم قادر به تشخیص آن نیست.
- توجه کنید که پس از حذف قوانین تولید تهی، تنها در صورتی می‌توانیم $A \stackrel{+}{\Rightarrow} A$ داشته باشیم که قوانین یکه به صورت $A \to B$ وجود داشته باشند.
- بنابراین همه قوانین تولید یکه به صورت $A \to B$ را نیز حذف می‌کنیم.
- الگوریتم‌های ساده‌سازی گرامرهای مستقل از متن برای حذف قوانین تولید تهی و یکه در مبحث نظریه زبان‌ها و ماشین‌ها بررسی می‌شوند.

حذف بازگشت چپ

- علت ایجاد اشکال در گرامر با وجود دورها به شرح زیر است.
- فرض کنید داشته باشیم $A_i \rightarrow A_j$ به طوری که $j > i$.
- در بازنویسی قانون $A_j \rightarrow A_i$ خواهیم داشت $A_j \rightarrow A_j$.
- اما برای حذف بازگشت چپ در این قانون با بازگشت چپ مواجه می‌شویم. در واقع اگر داشته باشیم $A \rightarrow A|\beta$ با حذف بازگشت چپ بلاواسطه به دست می‌آوریم $A \rightarrow \beta R, R \rightarrow R|\epsilon$ که همچنان بازگشتی چپ است.

حذف بازگشت چپ

- می‌خواهیم بازگشت چپ را در گرامر زیر حذف کنیم. قانون تولید تهی در اینجا مشکلی در اجرای الگوریتم ایجاد نمی‌کند، اما در حالت کلی قوانین تولید تهی را حذف می‌کنیم.

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

- متغیرها را به صورت S, A مرتب می‌کنیم. سپس قوانین زیر را تولید می‌کنیم.

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- با حذف بازگشت‌های چپ بلاواسطه گرامر زیر را به دست می‌آوریم.

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

فاکتورگیری چپ

- فاکتورگیری چپ روشی است برای تبدیل کردن یک گرامر به گرامری که برای تجزیه کنندهٔ بالا به پایین پیش‌بینی‌کننده مناسب باشد.
- وقتی برای جایگزین کردن یک متغیر با بدنهٔ قانون در فرایند اشتقاق دو انتخاب داشته باشیم، در مواردی می‌توانیم انتخاب را به تعویق بیاندازیم تا وقتی که ورودی بیشتری خوانده شود.
- برای مثال فرض کنید قوانین تولیدی به صورت زیر داریم.

$$\begin{array}{lcl} \mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\ & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \end{array}$$

- با خواندن توکن `if` از ورودی نمی‌توانیم تصمیم بگیریم کدام قانون را انتخاب کنیم.

فاکتورگیری چپ

- در حالت کلی اگر دو قانون $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ را داشته باشیم و ورودی α باشد، نمی‌توانیم تصمیم بگیریم کدام قانون را انتخاب کنیم، اما می‌توانیم گرامر را به گونه‌ای تغییر دهیم که انتخاب به تعویق بیافتد.
- می‌توانیم این گرامر را به صورت زیر بنویسیم :

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

فاکتورگیری چپ

- الگوریتم فاکتورگیری، گرامر G را دریافت می‌کند و گرامری تولید می‌کند که در آن فاکتورگیری چپ اعمال شده باشد.
- برای هر متغیر A ، بلندترین پیشوند α بین دو یا چند انتخاب را پیدا می‌کنیم. اگر $\alpha \neq \epsilon$ آنگاه قوانین $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$ را با قوانین زیر جایگزین می‌کنیم. قوانین γ قوانینی هستند که پیشوند آنها α نیست.

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

- این روند را برای همهٔ متغیرها تکرار می‌کنیم.

فاکتورگیری چپ

- گرامر زیر معادل گرامر `if-else` است. در این گرامر `i` و `t` و `e` نمایندهٔ `if` و `then` و `else` هستند.

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

- می‌توانیم این گرامر را به صورت زیر فاکتورگیری چپ کنیم.

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

- توجه کنید که هر دوی این گرامرها مبهم هستند.

ساختارهای غیرمستقل از متن

- برخی از ساختارها در زبان‌های برنامه‌نویسی را نمی‌توان توسط گرامرهای مستقل از متن توصیف کرد.
- برای مثال در بسیاری از زبان‌ها نیاز داریم که متغیر قبل از استفاده تعریف شده باشد.
- این ساختار را می‌توانیم به صورت wcw مدلسازی کنیم جایی که اولین w نماینده تعریف متغیر، c نماینده قسمتی از کد برنامه، و دومین w نماینده استفاده از متغیر باشد.
- می‌توان اثبات کرد که زبان $L = \{wcw \mid w \in (a|b)^*\}$ مستقل از متن نیست.

ساختارهای غیرمستقل از متن

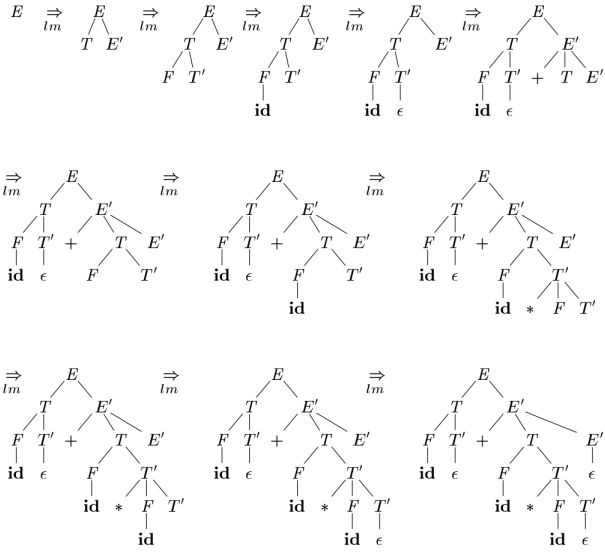
- در نتیجه نیاز به روش‌های دیگر برای تحلیل معنای برنامه‌ها داریم.
- یک مثال دیگر از ساختارهایی از زبان که مستقل از متن نیستند، به شرح زیر است. در زبان‌های برنامه‌نویسی نیاز است که تعداد آرگومان‌های ارسال شده به یک تابع برابر با تعداد پارامترهای تعریف شده در تابع باشد. فرض کنید تعریف دو تابع با n و m ورودی را به صورت a^n و b^m نشان دهیم و دو فراخوانی تابع از این دو تابع را به صورت c^n و d^m.
- این ساختار را با زبان $L = \{a^n b^m c^n d^m \mid n \geqslant 1, m \geqslant 1\}$ مدلسازی می‌کنیم. می‌توان اثبات کرد که این زبان مستقل از متن نیست.

تجزیه بالا به پایین

- تجزیهٔ بالا به پایین برای تجزیه یک رشته، درخت تجزیه را با شروع از ریشه می‌سازد.
- برای مثال برای تجزیه رشته $\mathtt{id} + \mathtt{id} * \mathtt{id}$ با استفاده از گرامر زیر، از تجزیه بالا به پایین صفحه بعد استفاده می‌کنیم.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

تجزیه بالا به پایین



تجزیه بالا به پایین

- ریشه درخت تجزیه متغیر آغازین است. در هرگام، تجزیه کننده باید تصمیم بگیرد از کدام یک از قوانین تولید استفاده کند برای اینکه بتواند رشته مورد نظر را تجزیه کند.
- ابتدا در مورد یک تجزیه کننده به نام تجزیه کننده کاهشی بازگشتی [1] صحبت می‌کنیم که در آن برای پیدا کردن قانون مناسب در فرایند تجزیه از پسگرد [2] استفاده می‌شود.
- سپس در مورد یک حالت خاص تجزیه کننده کاهشی بازگشتی به نام تجزیه کننده پیش بینی کننده [3] صحبت می‌کنیم که در آن به پسگرد نیازی نیست.

[1] recursive-descent parser
[2] backtrack
[3] predictive parser

تجزیه بالا به پایین

- تجزیه کننده پیش بینی کننده با بررسی چند نماد بعدی در رشته ورودی تصمیم می‌گیرد کدام قانون تولید را انتخاب کند و به پسگرد نیازی ندارد.
- گرامرهایی که با بررسی k نماد در ورودی می‌توانیم برای آنها تجزیه کننده پیش بینی کننده بسازیم، گرامرهای LL(k) نامیده می‌شوند.

تجزیه کننده کاهشی بازگشتی

- یک تجزیه کننده کاهشی بازگشتی برنامه‌ای است که از مجموعه‌ای از توابع تشکیل شده است به طوری که هر تابع متعلق به یکی از متغیرهای گرامر است. اجرای تجزیه کننده با فراخوانی تابع متعلق به متغیر آغازین شروع می‌شود و در نهایت اگر همهٔ رشته ورودی خوانده شد متوقف می‌شود.

تجزیه کننده کاهشی بازگشتی

- الگوریتم تجزیه کننده کاهشی بازگشتی در زیر نشان داده شده است.

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

تجزیه کننده کاهشی بازگشتی

- یک تجزیه کننده کاهشی بازگشتی با استفاده از یک الگوریتم پسگرد رشته ورودی را تجزیه می‌کند، اما برای تجزیه زبان‌های برنامه‌نویسی معمولا نیازی به پسگرد نیست.
- برای اینکه در تجزیه بالا به پایین از پسگرد استفاده کنیم، در خط (۱) برنامه قبل باید همه انتخاب‌های موجود برای جایگزینی متغیر A را امتحان کنیم. همچنین در خط (۷) در صورتی که به بن‌بست برخورد کردیم پیام خطا صادر نمی‌کنیم بلکه پسگرد انجام می‌شود.

تجزیه کننده کاهشی بازگشتی

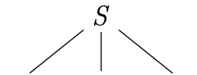
- گرامر زیر را در نظر بگیرید.

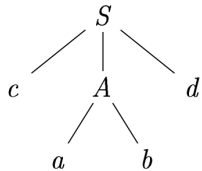
$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a$$

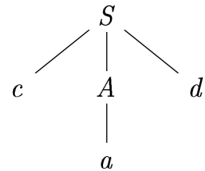
- برای ساختن یک درخت تجزیه از بالا به پایین برای رشته $w = cad$ با ریشه درخت تجزیه یعنی S آغاز می‌کنیم. S تنها یک قانون دارد. بنابراین رأس S را بسط می‌دهیم و فرزندان آن شامل c و A و d را می‌سازیم. اولین برگ یعنی c بر رشته ورودی منطبق می‌شود، پس در رشته ورودی جلو می‌رویم. برگ بعدی A است که یک متغیر با دو قانون است. آن را با استفاده از اولین قانون یعنی $A \rightarrow ab$ بسط می‌دهیم. نماد a در رشته ورودی بر دومین برگ یعنی a منطبق می‌شود پس در رشته ورودی به جلو می‌رویم. اما سومین برگ یعنی b بر نماد بعدی در ورودی یعنی d منطبق نمی‌شود پس باید به عقب برگردیم و یک قانون دیگر از A را انتخاب کنیم. با بازگشت به عقب باید در رشته ورودی هم به عقب برگردیم، پس در هر رأس باید اندیس رشته ورودی ذخیره شود.

تجزیه کننده کاهشی بازگشتی

- روند تجزیه کاهشی بازگشتی و پسگرد به صورت زیر است.







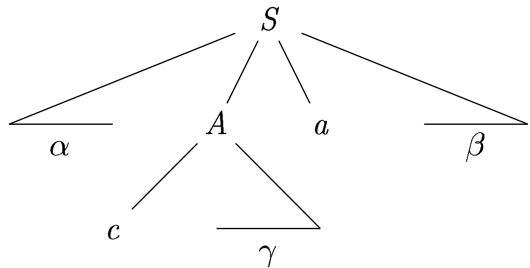
- اگر یک گرامر بازگشت چپ داشته باشد، تجزیه کننده کاهشی بازگشتی وارد حلقه بی‌پایان می‌شود.

توابع First و Follow

- برای ساختن تجزیه کننده‌های بالا به پایین و پایین به بالا به دو تابع مهم به نام First و Follow نیاز داریم که در اینجا به آنها اشاره می‌کنیم.
- در هنگام تجزیه بالا به پایین این توابع کمک می‌کنند قانون درست را با استفاده از نماد ورودی بعد انتخاب کنیم.
- در هنگام بازیابی خطا از توکن‌های تولید شده توسط تابع Follow استفاده می‌شود.

توابع First و Follow

- اگر α یک رشته از نمادهای گرامر باشد، آنگاه $\text{First}(\alpha)$ مجموعه‌ای از ترمینال‌هایی است که در ابتدای رشته‌های مشتق شده از α وجود دارند. اگر $\alpha \stackrel{*}{\Rightarrow} \epsilon$ آنگاه ϵ نیز در $\text{First}(\alpha)$ است.
- برای مثال در شکل زیر $A \stackrel{*}{\Rightarrow} c\gamma$ بنابراین c در $\text{First}(A)$ است.



توابع First و Follow

- تابع First در تجزیه پیش‌بینی کننده استفاده می‌شود. فرض کنید دو قانون $A \to \alpha|\beta$ را داشته باشیم و $First(\alpha)$ و $First(\beta)$ دو مجموعهٔ مجزا باشند. آنگاه با خواندن ورودی a می‌توانیم قانون مورد نظر برای اعمال را انتخاب کنیم زیرا a می‌تواند حداکثر در یکی از مجموعه‌های $First(\alpha)$ یا $First(\beta)$ باشد.

توابع First و Follow

- به ازای متغیر A تابع $\mathrm{Follow}(A)$ مجموعه ترمینال‌های a است که مستقیما در سمت راست متغیر A در یک صورت جمله‌ای در یک فرایند اشتقاق قرار می‌گیرند.
- به عبارت دیگر $\mathrm{Follow}(A)$ مجموعه ترمینال‌های a است که برای آنها اشتقاق $S \overset{*}{\Rightarrow} \alpha A a \beta$ وجود دارد. توجه کنید که بین A و a در فرایند اشتقاق می‌تواند متغیرهایی وجود داشته باشند ولی این متغیرها به تهی تبدیل می‌شوند.
- همچنین اگر A متغیر سمت راست باشد آنگاه $\$$ در $\mathrm{Follow}(A)$ قرار می‌گیرد. نماد $\$$ به معنای پایان رشته است و فرض می‌شود که این نماد در الفبا وجود ندارد.

توابع First و Follow

- برای محاسبه $First(X)$ برای نماد X قوانین زیر را اعمال می‌کنیم تا جایی که هیچ ترمینالی (یا رشتهٔ ϵ) نتواند به مجموعه $First(X)$ اضافه شود.

۱. اگر X یک ترمینال است آنگاه $First(X) = \{X\}$.

۲. اگر X یک متغیر است و $X \to Y_1 Y_2 \cdots Y_k$ یک قانون تولید است به ازای $k \geqslant 1$ آنگاه a در $First(X)$ قرار می‌گیرد اگر به ازای یک i دلخواه، ترمینال a در $First(Y_i)$ باشد و ϵ در همهٔ مجموعه‌های $First(Y_1), \cdots, First(Y_{i-1})$ باشد. در اینصورت خواهیم داشت $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. اگر به ازای همه $1 \leq j \leq k$، رشته ϵ در $First(Y_j)$ باشد آنگاه ϵ را به $First(X)$ اضافه می‌کنیم. برای مثال، هر نمادی در $First(Y_1)$ است در $First(X)$ نیز وجود دارد. اگر Y_1 رشته تهی ϵ را مشتق نمی‌کند آنگاه نماد دیگری به $First(X)$ اضافه نمی‌کنیم، اما اگر $Y_1 \stackrel{*}{\Rightarrow} \epsilon$ آنگاه $First(Y_2)$ را نیز به $First(X)$ می‌افزاییم و به همین ترتیب الی آخر.

۳. اگر $X \to \epsilon$ یک قانون تولید باشد، آنگاه ϵ را به $First(X)$ می‌افزاییم.

توابع First و Follow

- می‌توانیم $First(X_1X_2\cdots X_n)$ را به صورت زیر محاسبه کنیم. همهٔ نمادها غیر از ϵ از مجموعه $First(X_1)$ را به $First(X_1X_2\cdots X_n)$ می‌افزاییم. اگر ϵ در مجموعه $First(X_1)$ باشد، آنگاه همه نمادهای غیر ϵ از $First(X_2)$ را به $First(X_1X_2\cdots X_n)$ می‌افزاییم. اگر ϵ در $First(X_1)$ و $First(X_2)$ باشد آنگاه این همه نمادهای غیر از ϵ از $First(X_3)$ را به $First(X_1X_2\cdots X_n)$ می‌افزاییم. این روند را ادامه می‌دهیم. در نهایت اگر ϵ به ازای همه i ها در $First(X_i)$ باشد، آنگاه ϵ را به $First(X_1X_2\cdots X_n)$ اضافه می‌کنیم.

توابع First و Follow

- برای محاسبه $Follow(A)$ به ازای همه متغیرهای A قوانین زیر را اعمال می‌کنیم تا وقتی که هیچ نمادی نتواند به مجموعه $Follow(A)$ اضافه شود.

۱. اگر S متغیر آغازین باشد، نماد $ را در $Follow(S)$ اضافه می‌کنیم.

۲. اگر قانون $A \to \alpha B \beta$ وجود داشته باشد، آنگاه همهٔ نمادهای مجموعه $First(\beta)$ به جز رشته تهی ϵ را به $Follow(B)$ می‌افزاییم.

۳. اگر قانون $A \to \alpha B$ یا قانون $A \to \alpha B \beta$ وجود داشته باشد جایی که $First(\beta)$ حاوی ϵ باشد، آنگاه هر نمادی در $Follow(A)$ در $Follow(B)$ نیز قرار می‌گیرد.

توابع First و Follow

- برای مثال گرامر زیر را در نظر بگیرید.

$$S \to Am \mid An \mid kA \mid Bp$$
$$A \to qrB \mid s$$
$$B \to t$$

- با محاسبهٔ توابع First و Follow خواهیم داشت:

$$First(S) = \{ k, q, s, t\} \qquad Follow(S) = \{\$\}$$
$$First(A) = \{ q, s\} \qquad Follow(A) = \{\$, m, n\}$$
$$First(B) = \{ t\} \qquad Follow(B) = \{\$, m, n, p\}$$

توابع First و Follow

- گرامر زیر را در نظر بگیرید.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

توابع First و Follow

- با محاسبات توابع First و Follow به دست می‌آوریم :

$$\begin{aligned}
&\mathsf{First}(\mathsf{F}) = \mathsf{First}(\mathsf{T}) = \mathsf{First}(\mathsf{E}) = \{(, \mathrm{id}\} \\
&\mathsf{First}(\mathsf{E}') = \{+, \epsilon\} \\
&\mathsf{First}(\mathsf{T}') = \{*, \epsilon\} \\
&\mathsf{Follow}(\mathsf{E}) = \mathsf{Follow}(\mathsf{E}') = \{), \$\} \\
&\mathsf{Follow}(\mathsf{T}) = \mathsf{Follow}(\mathsf{T}') = \{+,), \$\} \\
&\mathsf{Follow}(\mathsf{F}) = \{+, *,), \$\}
\end{aligned}$$

گرامرهای LL(۱)

- تجزیه کننده‌های پیش‌بینی کننده یعنی تجزیه کننده‌های کاهشی بازگشتی که به پسگرد نیازی ندارند می‌توانند برای دسته‌ای از گرامرهای مستقل از متن به نام گرامرهای LL(۱) استفاده شوند.
- تجزیه کننده‌هایی که برای گرامرهای LL(۱) به کار می‌روند تجزیه کننده‌های LL(۱) نامیده می‌شوند. اولین L بدین معناست که خواندن ورودی از چپ به راست [1] انجام می‌شود و دومین L بدین معناست که تجزیه کننده اشتقاق چپ [2] تولید می‌کند و عدد ۱ بدین معناست که تجزیه کننده تنها یک نماد جلوتر [3] را در هر گام برای تصمیم‌گیری برای تجزیه بررسی می‌کند.
- دسته گرامرهای LL(۱) برای توصیف زبان‌های برنامه‌نویسی به اندازهٔ کافی توانمند است. البته در توصیف یک گرامر LL(۱) ملاحظاتی را باید در نظر گرفت. برای مثال گرامر نباید بازگشت چپ داشته باشد یا مبهم باشد.

[1] Left to right
[2] Leftmost derivation
[3] one input symbol of lookahead

گرامرهای LL(۱)

- گرامر مستقل از متن G یک گرامر LL(۱) است اگر و تنها اگر هنگامی که دو قانون مجزای $A \to \alpha|\beta$ وجود داشته باشند، شرط‌های زیر برقرار باشد.

۱. هیچ ترمینال a وجود ندارد به طوری که هردوی α و β رشته‌ای مشتق کنند، که هر دو با a آغاز شود.

۲. حداکثر یکی از صورت‌های جمله‌ای α و β می‌توانند رشته تهی تولید کنند.

۳. اگر $\beta \overset{*}{\Rightarrow} \epsilon$ آنگاه α هیچ رشته‌ای تولید نمی‌کند که با یک ترمینال در $\mathrm{Follow}(A)$ آغاز شود. به طور مشابه اگر $\alpha \overset{*}{\Rightarrow} \epsilon$ آنگاه β هیچ رشته‌ای تولید نمی‌کند که با یک ترمینال در $\mathrm{Follow}(A)$ آغاز شود.
فرض کنید این شرایط برقرار نباشد و داشته باشیم $S \overset{*}{\Rightarrow} Aa \Rightarrow \beta a \overset{*}{\Rightarrow} a$ و همچنین
$S \overset{*}{\Rightarrow} Aa \Rightarrow \alpha a \overset{*}{\Rightarrow} a\gamma a$. در اینصورت نمی‌توانیم تصمیم بگیریم با مشاهدهٔ توکن a در ورودی در فرایند اشتقاق برای متغیر A کدامیک از قوانین $A \to \alpha$ یا $A \to \beta$ را انتخاب کنیم.

گرامرهای LL(۱)

- شرط‌های اول و دوم معادل یکدیگرند. این دو شرط بدین معنی هستند که $First(\alpha)$ و $First(\beta)$ دو مجموعه مجزا هستند.
- شرط سوم بدین معنی است که اگر ϵ در $First(\beta)$ وجود داشت، آنگاه $First(\alpha)$ و $Follow(A)$ دو مجموعه مجزا هستند و به طور مشابه اگر ϵ در $First(\alpha)$ وجود داشت، $First(\beta)$ و $Follow(A)$ دو مجموعه مجزا هستند.

گرامرهای LL(۱)

- تجزیه کننده‌های پیش‌بینی کننده می‌توانند برای گرامرهای LL(۱) استفاده شوند زیرا انتخاب درست قانونی که می‌تواند در هرگام برای تجزیه به کار رود تنها با بررسی نماد بعدی در ورودی امکان پذیر است.
- برای مثال در گرامر زیر تنها با خواندن یکی از نمادهای `if` یا `while` یا `{` می‌توانیم تصمیم بگیریم کدام قانون را انتخاب کنیم.

```
stmt → if (expr) stmt else stmt
     | while (expr) stmt
     | { stmt-list }
```

الگوریتم تجزیه کننده پیش‌بینی کننده

- الگوریتم بعدی اطلاعاتی در مورد مجموعه‌های First و $Follow$ در یک جدول تجزیه پیش‌بینی کننده جمع‌آوری می‌کند. جدول تجزیه $M[A, a]$ یک آرایه دو بعدی است جایی که A یک متغیر و a یک ترمینال یا نماد $ است.
- الگوریتم بر پایه ایده زیر است : قانون $A \to \alpha$ انتخاب می‌شود اگر نماد بعدی a در $First(\alpha)$ باشد. تنها مشکل وقتی رخ می‌دهد که $\alpha = \epsilon$ یا $\alpha \stackrel{*}{\Rightarrow} \epsilon$ باشد. در اینصورت $A \to \alpha$ را انتخاب می‌کنیم اگر نماد ورودی در $Follow(A)$ باشد یا اگر به $ در رشته ورودی رسیده‌ایم و $ در $Follow(A)$ باشد.

الگوریتم تجزیه کننده پیش‌بینی کننده

- الگوریتم ساخت جدول تجزیه کننده پیش‌بینی کننده به صورت زیر است. این الگوریتم گرامر G را دریافت و جدول تجزیه M را تولید می‌کند.
- برای هر یک از قوانین $A \to \alpha$ از گرامر به صورت زیر عمل می‌کنیم.

۱. به ازای هریک از ترمینال‌های a در $\mathrm{First}(\alpha)$ قانون $A \to \alpha$ را به $M[A, a]$ اضافه می‌کنیم.

۲. اگر ϵ در $\mathrm{First}(\alpha)$ باشد، آنگاه به ازای هریک از ترمینال‌های a در $\mathrm{Follow}(A)$ قانون $A \to \alpha$ با به $M[A, a]$ اضافه می‌کنیم. اگر ϵ در $\mathrm{First}(\alpha)$ باشد و $\$$ در $\mathrm{Follow}(A)$ باشد آنگاه $A \to \alpha$ را به $M[A, \$]$ اضافه می‌کنیم.

- اگر پس از عملیات بالا هیچ قانونی در $M[A, a]$ قرار نگرفت، آنگاه در $M[A, a]$ مقدار خطا (error) قرار می‌دهیم. برای سادگی، خطاها را با خانه‌های خالی در جدول نمایش می‌دهیم.

الگوریتم تجزیه کننده پیش‌بینی کننده

- برای گرامر زیر جدول تجزیه زیر تولید می‌شود.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

الگوریتم تجزیه کننده پیش‌بینی کننده

- قانون $E \rightarrow TE'$ را در نظر بگیرید. از آنجایی که $First(TE') = First(T) = \{(, id\}$ این قانون به $M[E, (]$ و $M[E, id]$ افزوده شده است.
- قانون $E' \rightarrow +TE'$ به $M[E', +]$ افزوده شده است زیرا $First(+TE') = \{+\}$.
- از آنجایی که $Follow(E') = \{), \$\}$ قانون $E' \rightarrow \epsilon$ به $M[E',)]$ و $M[E', \$]$ افزوده شده است.

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

الگوریتم تجزیه کننده پیش‌بینی کننده

- الگوریتمی که شرح داده شد می‌تواند برروی هر گرامر G اعمال شود و یک جدول تجزیه M بسازد. برای هرگرامر LL(۱) یک جدول تجزیه وجود دارد که هر خانهٔ آن حاوی یک قانون تولید یا خطا است.
- برای برخی از گرامرها، جدول M ممکن است خانه‌ای داشته باشد که در آن بیش از یک قانون وجود دارد. اگر یک گرامر بازگشت چپ داشته باشد یا مبهم باشد، آنگاه M حداقل یک خانه با بیش از یک قانون دارد. با حذف بازگشت چپ و فاکتورگیری چپ در برخی موارد می‌توان یک گرامر را به یک گرامر LL(۱) تبدیل کرد. اما برای برخی از گرامرها معادل LL(۱) وجود ندارد.

الگوریتم تجزیه کننده پیش‌بینی کننده

- گرامر زیر و جدول تجزیه آن را در نظر بگیرید.

$$
\begin{aligned}
S &\rightarrow iEtSS' \mid a \\
S' &\rightarrow eS \mid \epsilon \\
E &\rightarrow b
\end{aligned}
$$

Non - terminal	Input Symbol					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

- در سلول $M[S', e]$ دو قانون $S' \rightarrow eS$ و $S' \rightarrow \epsilon$ قرار گرفته است.
- دلیل این امر این است که گرامر مبهم است.

تجزیه کننده پیش‌بینی کننده غیربازگشتی

- یک تجزیه کننده پیش‌بینی کننده غیربازگشتی [1] با نگهداری یک پشته به صورت صریح به جای استفاده از پشته فراخوانی ساخته می‌شود.
- این تجزیه کننده اشتقاق چپ را شبیه‌سازی می‌کند.
- اگر w رشته ورودی باشد که بر گرامر تطبیق داده شده باشد، آنگاه پشته یک دنباله از نمادهای α از گرامر را نگهداری می‌کند به طوری که $S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$.
- تجزیه کننده تشکیل شده است از یک بافر ورودی، یک پشته حاوی دنباله‌ای از نمادهای گرامر، یک جدول تجزیه که توسط الگوریتم قبل ساخته شده است و یک خروجی.

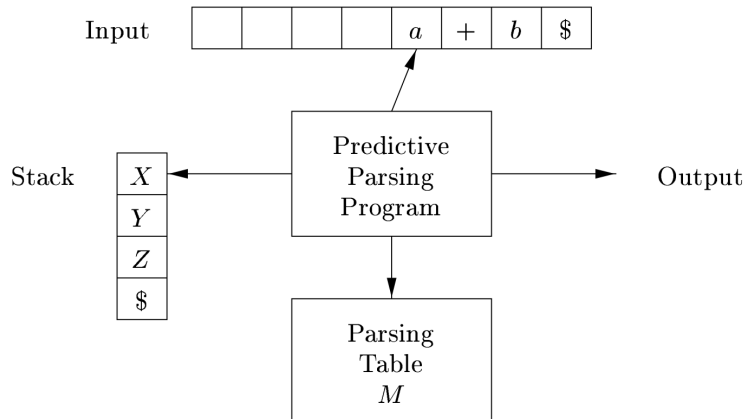
[1] nonrecursive predictive parser

تجزیه کننده پیش‌بینی کننده غیربازگشتی

- بافر ورودی رشته ورودی را دربرمی‌گیرد که با نماد $ ختم شده است. همچنین نماد $ انتهای پشته را نشان می‌دهد.
- تجزیه کننده نماد X را از روی پشته بر می‌دارد و نماد a را از ورودی می‌خواند. اگر X یک متغیر باشد، تجزیه کننده قانون تولیدی را که در $M[X, a]$ ذخیره شده انتخاب می‌کند. در غیراینصورت نماد X و نماد ورودی a باید تطبیق داده شوند.

تجزیه کننده پیش‌بینی کننده غیربازگشتی

- شمای این تجزیه کننده در زیر نشان داده شده است.



تجزیه کننده پیش‌بینی کننده غیربازگشتی

- الگوریتم رشته w و جدول M برای گرامر G را دریافت می‌کند. اگر w در $L(G)$ باشد یک اشتقاق چپ برای w تولید می‌کند در غیراینصورت پیام خطا صادر می‌کند.
- در ابتدا تجزیه کننده در پیکربندی $w\$$ [1] قرار دارد و نماد آغازین S در پشته برروی نماد انتهای پشته یعنی $ قرار گرفته می‌شود.

[1] configuration

تجزیه کننده پیش‌بینی کننده غیربازگشتی

- الگوریتم زیر عملیات تجزیه پیش‌بینی کننده را نشان می‌دهد.

let a be the first symbol of w;
let X be the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if ($X = a$) pop the stack and **let** a be the next symbol of w;
    else if (X is a terminal) *error*();
    else if ($M[X, a]$ is an error entry) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
    }
    let X be the top stack symbol;
}

تجزیه کننده پیش‌بینی کننده غیربازگشتی

- گرامر زیر را در نظر بگیرید.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

تجزیه کننده پیش‌بینی کننده غیربازگشتی

- جدول تجزیه این گرامر را قبلا به صورت زیر محاسبه کردیم.

NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	$($	$)$	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

تجزیه کننده پیش‌بینی کننده غیربازگشتی

- با دریافت ورودی $\mathtt{id} + \mathtt{id} * \mathtt{id}$ تجزیه کننده پیش‌بینی کننده غیربازگشتی یک فرایند اشتقاق چپ به صورت زیر تولید می‌کند.

$$\mathtt{E} \underset{lm}{\Rightarrow} \mathtt{TE'} \underset{lm}{\Rightarrow} \mathtt{FT'E'} \underset{lm}{\Rightarrow} \mathtt{idT'E'} \underset{lm}{\Rightarrow} \mathtt{idE'} \underset{lm}{\Rightarrow} \mathtt{id} + \mathtt{TE'} \underset{lm}{\Rightarrow} \cdots$$

تجزیه کننده پیش‌بینی کننده غیربازگشتی

- این فرایند اشتقاق به صورت زیر تولید می‌شود.

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

تجزیه کننده پیش‌بینی کننده غیربازگشتی

- یک صورت جمله‌ای در فرایند اشتقاق متناظر است با ورودی تطبیق داده شده (در ستون Matched)که به دنبال آن محتوای پشته قرار داده شده است.

بازیابی خطا در تجزیه کننده پیش‌بینی کننده

- یک خطا در تجزیه پیش‌بینی کننده رخ می‌دهد وقتی که یک ترمینال برروی پشته برروی نماد ورودی منطبق نشود و یا وقتی که با خواندن متغیر A از پشته و نماد a از رشته ورودی، $M[A, a]$ یک خطا باشد یا به عبارت دیگر خانهٔ $M[A, a]$ در جدول تجزیه خالی باشد.

بازیابی خطا با توکن همگام‌کننده

- بازیابی خطا با توکن همگام‌کننده [1] بر این پایه است که در هنگام رخداد خطا از نمادهای ورودی چشم‌پوشی شود تا جایی که یک توکن همگام‌کننده [2] پیدا شود.
- مجموعه توکن‌های همگام‌کننده باید به نحوی انتخاب شود که تجزیه‌کننده بتواند به سرعت خطا را بازیابی کند.

[1] panic mode error recovery
[2] synchronizing token

بازیابی خطا با توکن همگام‌کننده

- موارد زیر برای انتخاب توکن‌های همگام‌کننده می‌توانند استفاده شوند.

۱. همهٔ نمادها در $\mathrm{Follow}(A)$ را در مجموعه همگام کننده متغیر A قرار می‌دهیم. اگر از همهٔ توکن‌ها چشم‌پوشی کنیم تا یکی از اعضای $\mathrm{Follow}(A)$ مشاهده شود و A از پشته خارج شود، به احتمال زیاد تجزیه می‌تواند ادامه پیدا کند.

- برای مثال فرض کنید قوانین $S \to \mathrm{E};S|\epsilon$ و $\mathrm{E} \to \mathrm{T}\mathrm{E}'$ و $\mathrm{E}' \to +\mathrm{T}\mathrm{E}'|\epsilon$ در یک گرامر وجود داشته باشد. اگر در پشته مقدار $\mathrm{E}';\mathrm{S}\$$ وجود داشته باشد و به عبارت $\mathrm{id}(\mathrm{id}+\mathrm{id};$ برخورد کنیم، از کاراکترها چشم‌پوشی می‌شود تا اینکه به یک کاراکتر نقطه ویرگول برخورد کنیم، زیرا $\mathrm{Follow}(\mathrm{E}') = \{;\}$.

بازیابی خطا با توکن همگام‌کننده

۲. تنها اعضای $\mathrm{Follow}(A)$ برای مجموعه همگام‌کننده A کافی نیستند. برای مثال اگر دستورات با نقطه ویرگول خاتمه پیدا کنند، آنگاه کلمه‌های کلیدی که در ابتدای دستورات بعدی هستند در مجموعه Follow قرار نمی‌گیرند. بنابراین اگر یک نقطه ویرگول جا افتاده باشد، از کلمات کلیدی دستورات بعدی چشم‌پوشی می‌شود. معمولاً در زبان‌های برنامه‌نویسی یک ساختار سلسله مراتبی وجود دارد. برای مثال عبارات در دستورات استفاده می‌شوند و دستورات در بلوک‌ها و الی آخر. می‌توانیم نمادهایی را که متغیرهای سلسله‌مراتب بالاتر با آنها آغاز می‌شوند، به مجموعه همگام‌کننده از متغیرهای سلسله مراتب پایین‌تر اضافه کنیم. برای مثال، می‌توانیم کلمات کلیدی را که دستورات با آنها شروع می‌شوند در مجموعه‌های همگام‌کننده متغیرهایی قرار دهیم که عبارات را تولید می‌کنند.

- برای مثال فرض کنید قوانین $S \to E;S|\epsilon$ و $E \to TE'|\ \mathtt{int\ id}$ و $E' \to +TE'|\epsilon$ در یک گرامر وجود داشته باشد. از آنجایی که E می‌تواند با کلمه $\mathtt{int}$ آغاز شود، آن را به مجموع اگر در پشته مقدار $E';S\$$ وجود داشته باشد و به عبارت $\mathtt{id}(\mathtt{id} + \mathtt{id}\ \ \mathtt{int\ id};$ برخورد کنیم، از کاراکترها چشم‌پوشی می‌شود تا اینکه به یک کاراکتر نقطه ویرگول یا توکن $\mathtt{int}$ برخورد کنیم، زیرا $\mathrm{synch}(E') = \{;, \mathtt{int}\}$.

بازیابی خطا با توکن همگام‌کننده

۳. اگر نمادهای $\mathrm{First}(A)$ را به مجموعه همگام‌کننده متغیر A اضافه کنیم، آنگاه می‌توانیم تجزیه را با توجه به متغیر A ادامه دهیم اگر یک نماد در $\mathrm{First}(A)$ در ورودی ظاهر شود.

- برای مثال فرض کنید قوانین $S \to E;S|\epsilon$ و $E \to TE'$ و $E' \to +TE'|\epsilon$ در یک گرامر وجود داشته باشد. اگر در پشته مقدار $E';S\$$ وجود داشته باشد و به عبارت $\mathrm{id};(+\mathrm{id})))$ برخورد کنیم، از کاراکترهای))) چشم‌پوشی می‌شود و یک پیام خطا صادر می‌شود تا اینکه به یک کاراکتر $+$ برخورد کنیم، زیرا $\mathrm{First}(E') = \{+\}$.

بازیابی خطا با توکن همگام‌کننده

۴. اگر یک متغیر بتواند رشته تهی تولید کند، آنگاه قانون تولیدی که به رشته تهی می‌انجامد می‌تواند به عنوان پیش فرض در نظر گرفته شود. با این کار تشخیص خطا به تأخیر می‌افتد اما از طرفی باعث می‌شود خطاها از دست نروند.

- برای مثال فرض کنید قوانین $S \to E; S | \epsilon$ و $E \to TE'$ و $E' \to +TE' | \epsilon$ در یک گرامر وجود داشته باشد. اگر در پشته مقدار $E'; S\$$ وجود داشته باشد و به عبارت $id(((+id;$ برخورد کنیم، متغیر E' به تهی تبدیل می‌شود و از کاراکترهای ورودی چشم‌پوشی می‌شود تا کاراکتر نقطه ویرگول مشاهده شود.

بازیابی خطا با توکن همگام‌کننده

۵. اگر یک ترمینال برروی پشته باشد که نتواند تطبیق داده شود، یک ایده این است که ترمینال از روی پشته برداشته شود و خطایی صادر شود مبنی براینکه ترمینال توسط کامپایلر اضافه شده است و عملیات تجزیه ادامه پیدا کند.

- برای مثال فرض کنید قوانین $S \to E; S|\epsilon$ و $E \to TE'$ و $E' \to +TE'|\epsilon$ در یک گرامر وجود داشته باشد. اگر در پشته مقدار $E'; S\$$ وجود داشته باشد و به عبارت $\texttt{id id};$ برخورد کنیم، توکن $+$ اضافه می‌شود و یک پیام خطا صادر می‌شود، مبنی بر اینکه توکن $+$ در ورودی فراموش شده است، و تجزیه ادامه پیدا می‌کند.

بازیابی خطا با توکن همگام‌کننده

- گرامر زیر و جدول تجزیه متناظر با آن را در نظر بگیرید.

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

بازیابی خطا با توکن همگام‌کننده

- در جدول زیر واژه synch معادل با توکن‌های همگام‌کننده‌ای است از مجموعه $Follow$ برای هر یک از متغیرها استخراج شده است.

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	synch	synch	$F \to (E)$	synch	synch

بازیابی خطا با توکن همگام‌کننده

- از این جدول به شرح زیر استفاده می‌شود. اگر تجزیه کننده به سلول $M[A, a]$ رسید که خالی بود آنگاه از a چشم‌پوشی می‌شود. اگر تجزیه کننده به کلمه synch برخورد کرد، متغیر از روی پشته برداشته می‌شود تا تجزیه بتواند ادامه پیدا کند. اگر یک توکن از روی پشته بر نماد ورودی تطبیق داده نشود، آنگاه توکن از پشته برداشته می‌شود.

بازیابی خطا با توکن همگام‌کننده

- با خواندن ورودی $\texttt{id} * +\texttt{id}*$ تجزیه کننده به صورت زیر عمل می‌کند.

STACK	INPUT	REMARK
$E\,\$$	$* \mathbf{id} * + \mathbf{id}\,\$$	error, skip $*$
$E\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	**id** is in $\text{FIRST}(E)$
$TE'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$T'E'\,\$$	$* + \mathbf{id}\,\$$	
$* FT'E'\,\$$	$* + \mathbf{id}\,\$$	
$FT'E'\,\$$	$+ \mathbf{id}\,\$$	error, $M[F,+] = \text{synch}$
$T'E'\,\$$	$+ \mathbf{id}\,\$$	F has been popped
$E'\,\$$	$+ \mathbf{id}\,\$$	
$+ TE'\,\$$	$+ \mathbf{id}\,\$$	
$TE'\,\$$	$\mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id}\,\$$	
$\mathbf{id}\,T'E'\,\$$	$\mathbf{id}\,\$$	
$T'E'\,\$$	$\$$	
$E'\,\$$	$\$$	
$\$$	$\$$	

بازیابی خطا با توکن همگام‌کننده

- معمولاً یک کامپایلر خوب پیام‌های خطایی صادر می‌کند که اطلاعات مفیدی به دست برنامه‌نویس می‌دهد.

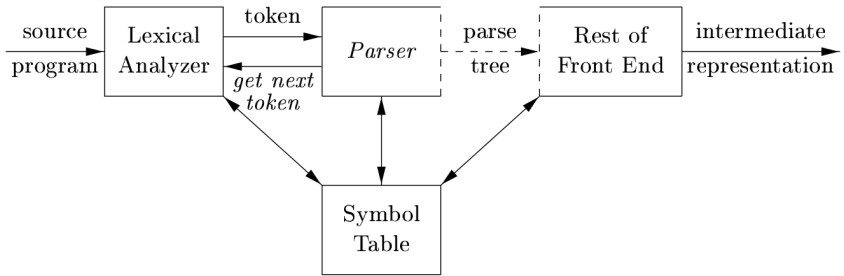
بازیابی خطا با جایگزینی توکن‌ها

- بازیابی خطا با جایگزینی توکن‌ها [1] بدین صورت پیاده‌سازی می‌شود که به جای سلول‌های خالی در جدول تجزیه، توابعی قرار می‌گیرند که بازیابی خطا را انجام می‌دهند. این توابع می‌توانند نمادهایی را تغییر دهند یا اضافه کنند و یا حذف کنند و پیام خطای مناسب صادر کنند. همچنین این توابع می‌توانند از پشته نمادهایی را خارج کنند یا نمادهایی را جایگزین کنند و یا نمادهایی را به پشته اضافه کنند. باید اطمینان حاصل شود که این توابع ایجاد حلقه بی‌پایان نمی‌کنند.

[1] phrase-level error recovery

تجزیه پایین به بالا

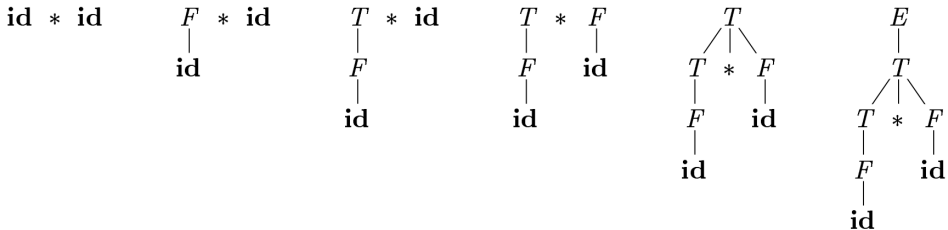
- یک تجزیه کننده پایین به بالا [1] درخت تجزیه برای یک ورودی را از برگ‌ها (پایین) به سمت ریشه (بالا) می‌سازد.
- فرض کنید می‌خواهیم رشته `id * id` را با استفاده از یک تجزیه کننده پایین به بالا برای گرامر زیر تجزیه کنیم.



[1] bottom-up parser

تجزیه پایین به بالا

- فرایند اشتقاق و ساخت درخت تجزیه از پایین به بالا برای این رشته به صورت زیر خواهد بود.



تجزیه پایین به بالا

- در این قسمت یک روش کلی برای تجزیه پایین به بالا به نام تجزیه انتقال کاهش [1] معرفی می‌کنیم.
- یکی از دسته‌های مهم گرامرها که برای آنها تجزیه کننده انتقال کاهش می‌تواند ساخته شود، دسته گرامرهای LR نامیده می‌شود.

[1] shift-reduce parsing

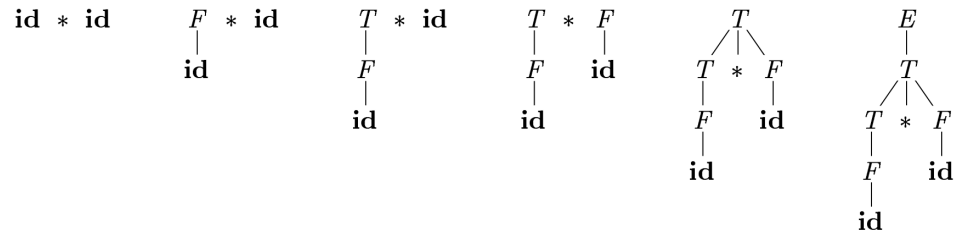
کاهش

- یک تجزیه کننده پایین به بالا با دریافت یک رشته ورودی آن را به متغیر آغازین کاهش می‌دهد. در هرگام کاهش [1] ، یک زیر رشته از ورودی بر بدنهٔ یک قانون تولید تطبیق پیدا می‌کند و به متغیر آن قانون تولید کاهش پیدا می‌کند. یک تجزیه کننده پایین به بالا تعیین میکند کدام قسمت از رشته ورودی توسط کدام یک از قوانین تولید کاهش پیدا کند.

[1] reduction

کاهش

- در شکل زیر id ∗ id به F ∗ id کاهش پیدا می‌کند و سپس به ترتیب به T ∗ id ، T ∗ F ، T و در نهایت رشته ورودی به E کاهش پیدا می‌کند.



- در گام اول برای کاهش از قانون F → id استفاده می‌شود. در برخی از گام‌ها چند انتخاب برای کاهش وجود دارد که تجزیه کنند باید تصمیم بگیرید از کدام قانون و کدام زیر رشته برای کاهش استفاده کند.

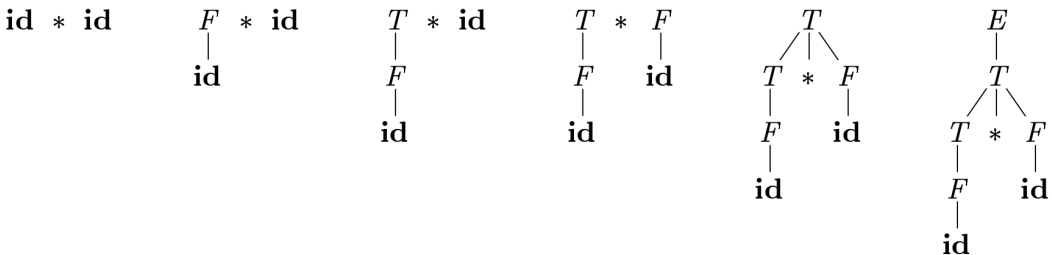
کاهش

- فرایند کاهش معکوس فرایند اشتقاق است. در فرایند اشتقاق یک متغیر در یک صورت جمله‌ای با بدنه یک قانون از آن متغیر جایگزین می‌شود. اما در فرایند کاهش یک زیررشته از صورت جمله‌ای بر بدنهٔ یک قانون منطبق و با متغیر متعلق به آن قانون جایگزین می‌شود. بنابراین تجزیه کننده پایین به بالا یک اشتقاق به صورت معکوس می‌سازد.

کاهش

- در شکل زیر فرایند اشتقاق راست

$$\mathtt{E} \Rightarrow \mathtt{T} \Rightarrow \mathtt{T}*\mathtt{F} \Rightarrow \mathtt{T}*\mathtt{id} \Rightarrow \mathtt{F}*\mathtt{id} \Rightarrow \mathtt{id}*\mathtt{id}$$



کاهش

- در تجزیه پایین به بالا ورودی از چپ به راست خوانده می‌شود و یک اشتقاق راست [1] به صورت معکوس تولید می‌شود.
- یک هندل [2] زیر رشته است که بر بدنه یکی از قوانین تولید تطبیق داده می‌شود. کاهش یک هندل به معنای جایگزین کردن آن با متغیر قانون تولید انتخاب شده است. کاهش هندل یک گام در فرایند اشتقاق راست معکوس است.

[1] rightmost derivation
[2] handle

کاهش

- برای مثال در جدول زیر هندل در هر گام از فرایند کاهش مشخص شده است. برای خوانایی بیشتر و تمیز دادن توکن‌های `id` از اندیس استفاده شده است.

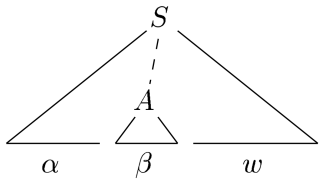
Right Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

کاهش

- در این فرایند کاهش در گام سوم گرچه T در $\mathtt{T} * \mathtt{id}_2$ می‌تواند به عنوان هندل انتخاب شود چرا که قانون $\mathtt{E} \longrightarrow \mathtt{T}$ وجود دارد، اما T به عنوان هندل انتخاب نمی‌شود چرا که در اینصورت متغیر آغازین E به دست نمی‌آید و رشته تجزیه نمی‌شود.

کاهش

- اگر داشته باشیم $\alpha\beta w \underset{rm}{\Rightarrow} \alpha Aw$ آنگاه قانون تولید $A \to \beta$ یک هندل برای صورت جمله‌ای $\alpha\beta w$ به دنبال α در فرایند کاهش است.



- توجه کنید که در تعریف بالا w تنها از ترمینال‌ها تشکیل شده است. برای سادگی به جای $A \to \beta$ می‌گوییم β یک هندل برای $\alpha\beta w$ است.
- اگر یک گرامر مبهم باشد، چند هندل در فرایند کاهش وجود خواهد داشت.

کاهش

- فرایند اشتقاق راست معکوس با کاهش هندل به دست می‌آید. در این فرایند با یک رشته w از ترمینال‌ها آغاز می‌کنیم. اگر w رشته از گرامر باشد، آنگاه $w = \gamma_n$ جایی که γ_n برابراست با n امین صورت جمله‌ای در فرایند اشتقاق راست.

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

- برای ساختن این اشتقاق به صورت معکوس، هندل γ_n در γ_n پیدا می‌شود و با A_n با استفاده از قانون $A_n \rightarrow \beta_n$ جایگزین می‌شود تا صورت جمله γ_{n-1} به دست بیاید.
- الگوریتم تجزیه پایین به بالا روشی برای یافتن هندل توصیف می‌کند. این فرایند ادامه پیدا می‌کند تا در نهایت متغیر S به دست بیاید. در اینصورت رشته تجزیه شده است و متعلق به گرامر است.

تجزیه انتقال‌کاهش

- تجزیه انتقال‌کاهش نوعی تجزیه پایین به بالا است که در آن یک پشته نمادهای گرامر را نگهداری می‌کند و در بافر ورودی باقیمانده رشته ورودی برای تجزیه مشخص شده‌است.
- هندل همیشه برروی پشته قرار گرفته است.
- در انتهای پشته و همچنین در انتهای رشته ورودی علامت $ را قرار می‌دهیم.
- در ابتدای وضعیت پشته و رشته ورودی به صورت زیر است.

STACK	INPUT
$	w $

تجزیه انتقال‌کاهش

- تجزیه کننده ورودی را از چپ به راست می‌خواند و تعداد صفر یا بیشتر نماد از ورودی را در پشته قرار می‌دهد تا وقتی که یک هندل β برروی پشته برای کاهش یافت شود. سپس β از پشته حذف می‌شود و با متغیر قانونی که کاهش با استفاده از آن انجام می‌شود جایگزین می‌شود.

- این فرایند ادامه پیدا می‌کند تا اینکه یا تجزیه کننده با خطا روبرو شود و یا در پشته متغیر آغازین قرار بگیرد و ورودی به پایین برسد. در اینصورت وضعیت پشته و رشته ورودی به صورت زیر است و رشته ورودی به درستی تجزیه شده است.

STACK	INPUT
$\$ S$	$\$$

تجزیه انتقال‌کاهش

- شکل زیر گام‌های یک تجزیه کننده انتقال‌کاهش را برای تجزیه رشته `id * id` نشان می‌دهد.

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \to T * F$
$\$ T$	$\$$	reduce by $E \to T$
$\$ E$	$\$$	accept

تجزیه انتقال‌کاهش

- در فرایند تجزیه انتقال‌کاهش چهار عملیات می‌تواند توسط تجزیه کننده اجرا شود.

۱. انتقال [1] : یک نماد از ورودی به روی پشته انتقال پیدا می‌کند.

۲. کاهش [2] : یک هندل که نماد سمت راست آن به بالای پشته است و نماد سمت چپ آن در پشته قرار دارد مشخص می‌شود و با استفاده از یک قانون گرامر کاهش پیدا می‌کند. هندل از پشته حذف و متغیر مربوط برروی پشته اضافه می‌شود. هندل همیشه بالای پشته قرار می‌گیرد نه در وسط آن.

۳. پذیرش [3] : عملیات تجزیه به اتمام رسیده و رشته پذیرفته شده است.

۴. خطا [4] : یک خطا نحوی تشخیص داده شده و یک تابع بازیابی خطا فراخوانی می‌شود.

---

[1] shift
[2] reduce
[3] accept
[4] error

ناسازگاری در تجزیه انتقال‌کاهش

- برای برخی از گرامرهای مستقل از متن تجزیه انتقال‌کاهش نمی‌تواند استفاده شود. در چنین گرامرهایی تجزیه کننده انتقال‌کاهش به یک پیکربندی می‌رسد که در آن تجزیه کننده نمی‌تواند تصمیم بگیرد عملیات انتقال انجام دهد و یا عملیات کاهش. به این شرایط ناسازگاری انتقال‌کاهش [1] گفته می‌شود. همچنین ممکن است تجزیه‌کننده نتواند تصمیم بگیرد از بین چند کاهش کدام یک را اعمال کند. به این شرایط ناسازگاری کاهش‌کاهش [2] گفته می‌شود.

[1] shift/reduce conflict
[2] reduce/reduce conflict

ناسازگاری در تجزیه انتقال‌کاهش

- اگر یک تجزیه کننده انتقال‌کاهش با آگاهی از k نماد جلویی در ورودی و آگاهی از محتوای پشته بتواند تصمیم بگیرد انتقال را اعمال کند و یا کاهش و بتواند تصمیم درستی درمورد انتخاب عملیات کاهش بگیرد گرامر مورد تجزیه یک گرامر LR(k) نامیده می‌شود.
- گرامری که تجزیه‌کننده انتقال‌کاهش برای تجزیه جملات با آن تنها نیاز به آگاهی از یک نماد جلویی در ورودی داشته باشد، یک گرامر LR(1) نامیده می‌شود.
- حرف L بدین معنی است که ورودی از چپ به راست [1] خوانده می‌شود و حرف R بدین معنا است که یک اشتقاق راست به صورت معکوس [2] ایجاد می‌شود و k بدین معنی است که آگاهی از k نماد جلویی از ورودی برای تجزیه جمله کافی است.

[1] left-to-right
[2] rightmost derivation in reverse

ناسازگاری در تجزیه انتقال‌کاهش

- یک گرامر مبهم هیچ‌گاه نمی‌تواند LR باشد.
- گرامر زیر را در نظر بگیرید.

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

- اگر پیکربندی زیر را در هنگام تجزیه داشته باشیم، نمی‌توانیم تصمیم بگیریم آیا `ifexprthenstmt` هندل است یا خیر.

STACK	INPUT
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots$ \$

- در اینجا یک ناسازگاری انتقال‌کاهش به وجود می‌آید.
- تجزیه انتقال‌کاهش می‌تواند با کمی تغییرات برای گرامرهای مبهم نیز استفاده شود.

ناسازگاری در تجزیه انتقال‌کاهش

- در برخی مواقع تجزیه‌کننده نمی‌تواند تصمیم بگیرد از بین چند هندل کدام یک را انتخاب کند و کدام قانون تولید را در فرآیند کاهش اعمال کند.

- فرض کنید گرامری به صورت زیر داریم که در آن فراخوانی تابع و تعریف آرایه شبیه به یکدیگر تعریف می‌شوند و هردو از نماد پرانتز استفاده می‌کنند

$$
\begin{array}{lrcl}
(1) & \mathit{stmt} & \rightarrow & \textbf{id}\ (\ \mathit{parameter\_list}\ ) \\
(2) & \mathit{stmt} & \rightarrow & \mathit{expr}\ :=\ \mathit{expr} \\
(3) & \mathit{parameter\_list} & \rightarrow & \mathit{parameter\_list}\ ,\ \mathit{parameter} \\
(4) & \mathit{parameter\_list} & \rightarrow & \mathit{parameter} \\
(5) & \mathit{parameter} & \rightarrow & \textbf{id} \\
(6) & \mathit{expr} & \rightarrow & \textbf{id}\ (\ \mathit{expr\_list}\ ) \\
(7) & \mathit{expr} & \rightarrow & \textbf{id} \\
(8) & \mathit{expr\_list} & \rightarrow & \mathit{expr\_list}\ ,\ \mathit{expr} \\
(9) & \mathit{expr\_list} & \rightarrow & \mathit{expr}
\end{array}
$$

ناسازگاری در تجزیه انتقال‌کاهش

- حال یک عبارت به صورت `p(i, j)` در ورودی پس از تحلیل لغوی به صورت `id(id, id)` تبدیل می‌شود و به تجزیه‌کننده تحویل داده می‌شود. پس از انتقال سه توکن برروی پشته، تجزیه کننده انتقال‌کاهش در وضعیت زیر قرار می‌گیرد.

STACK	INPUT
$\cdots$ **id (id**	**, id**) $\cdots$

- در اینجا دو قانون ۵ برای کاهش `p` به نام تابع و قانون ۷ برای کاهش `p` به نام آرایه می‌تواند استفاده شوند.
- یک راه‌حل این است که به جای `id` در قانون تولید ۱ از توکن `procid` استفاده شود. این راه حل تحلیل‌گر لغوی را پیچیده می‌کند. زیرا تحلیل‌گر نیاز به استفاده از جدول علائم خواهد داشت.
- یک راه‌حل دیگر تغییر ساختار نحوی برنامه و تغییر زبان برنامه‌نویسی است.

تجزیه LR ساده

- بسیاری از تجزیه‌کننده‌های پایین به بالا بر مبنای تجزیه LR(k) هستند.
- حرف L بدین معناست که ورودی از چپ به راست [1] خوانده می‌شود و حرف R بدین معناست که تجزیه با استفاده از یک فرایند اشتقاق راست معکوس [2] انجام می‌شود و k به معنای تعداد نمادهای جلویی است که در برای تصمیم‌گیری در فرایند تجزیه استفاده می‌شود.
- برای تجزیه زبان‌های برنامه‌نویسی معمولاً از تجزیه‌کننده‌های LR(1) و LR(0) استفاده می‌شود.
- وقتی از تجزیه‌کننده LR صحبت می‌کنیم منظور تجزیه‌کننده LR(1) است.
- ابتدا مورد یک تجزیه‌کننده LR ساده صحبت می‌کنیم و سپس با روش‌های پیچیده‌تر از جمله تجزیه‌کننده LR استاندارد [3] و تجزیه‌کننده LALR آشنا می‌شویم.

[1] left-to-right
[2] rightmost derivation in reverse
[3] canonical LR

تجزیه LR ساده

- تجزیه‌کننده‌های LR شبیه به تجزیه‌کننده‌ها LL از یک جدول تجزیه استفاده می‌کنند.
- گرامرهایی که می‌توان برای آنها یک تجزیه‌کننده LR طراحی کرد، گرامرهای LR نامیده می‌شوند.

تجزیه LR ساده

- تجزیه LR به چند دلیل پرکاربرد است :

۱. تجزیه‌کننده‌های LR می‌توانند همهٔ ساختارهای زبان‌های برنامه‌نویسی را که برای آنها یک گرامر مستقل از متن وجود دارد تجزیه کنند. گرامرهای مستقل از متنی وجود دارند که LR نیستند اما این گرامرها در زبان‌های برنامه‌نویسی استفاده نمی‌شوند.

۲. تجزیه‌کنند LR روشی است غیربازگشتی برای پیاده‌سازی تجزیه انتقال‌کاهش و در عین حال به اندازه بقیه روش‌های تجزیه کاراست.

۳. تجزیه‌کننده LR خطاهای نحوی را با خواندن رشته از چپ به راست به سرعت تشخیص می‌دهد.

۴. دسته گرامرهای LR ابر مجموعه دسته گرامرهای LL است.

- تنها عیب تجزیه‌کننده LR این است که ساختن آن بسیار پیچیده است.

ماشین LR(0)

- چگونه یک تجزیه‌کننده انتقال‌کاهش تشخیص می‌دهد چه زمانی انتقال و چه زمانی کاهش انجام دهد؟
- برای مثال اگر محتوای پشته $T باشد و نماد بعدی $*$ باشد، تجزیه‌کننده چگونه تشخیص می‌دهد T هندل نیست و باید به جای کاهش انتقال انجام دهد؟

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \to T * F$
$\$ T$	$\$$	reduce by $E \to T$
$\$ E$	$\$$	accept

ماشین LR(0)

- یک تجزیه‌کننده LR تصمیم انتقال یا کاهش را با نگهداری تعدادی حالت انجام می‌دهد.
- این حالت‌ها نمایندهٔ مجموعه‌ای از آیتم‌ها [1] هستند.
- یک آیتم LR(0) از گرامر G یک قانون تولید گرامر G است که تعدادی نقطه در بین نمادهای بدنه آن افزوده شده‌اند.
- بنابراین قانون $A \to XYZ$ چهار آیتم به صورت زیر دارد :

$$A \to \cdot XYZ$$
$$A \to X \cdot YZ$$
$$A \to XY \cdot Z$$
$$A \to XYZ \cdot$$

- قانون تولید $A \to \epsilon$ تنها یک آیتم به صورت $A \to \cdot$ دارد.

[1] item

ماشین LR(0)

- به طور شهودی، یک آیتم نشان می‌دهد چه مقداری از یک قانون تولید در هر لحظه دیده شده است.
- برای مثال آیتم $A \to \cdot XYZ$ نشان دهنده این است که می‌توانیم رشته ورودی را از XYZ مشتق کنیم.
- آیتم $A \to X \cdot YZ$ نشان دهنده این است که ورودی خوانده شده از X مشتق شده و ممکن است بتوانیم ادامه رشته را از YZ مشتق کنیم.
- آیتم $A \to XYZ\cdot$ نشان دهنده این است که رشته خوانده شده از XYZ مشتق شده و می‌توانیم XYZ را به A دهیم.

ماشین LR(0)

- یک گروه از مجموعه‌های آیتم‌های LR(0) یک گروه LR(0) استاندارد نامیده می‌شود که از آن برای ساختن یک ماشین متناهی قطعی برای تصمیم‌گیری در فرایند ترجمه استفاده می‌شود.
- این ماشین را ماشین LR(0) [1] می‌نامیم.

[1] LR(۰) automaton

ماشین LR(0)

- هر حالت از ماشین LR(0) نشان دهنده مجموعه‌ای از آیتم‌ها در گروه LR(0) استاندارد [1] است.
- برای ساختن گروه LR(0) استاندارد برای یک گرامر، یک گرامر با افزودن دو تابع `Closure` و `Goto` می‌سازیم.

[1] canonical LR(۰) collection

ماشین LR(0)

- اگر G یک گرامر با نماد آغازین S باشد، آنگاه G یک گرامر افزوده شده [1] برای G است که نماد آغازین S' و قانون $S' \rightarrow S$ در آن افزوده شده است.

[1] augmented grammar

تابع Closure

- اگر I مجموعه‌ای از آیتم‌ها بر روی گرامر G باشد، آنگاه closure(I) مجموعه‌ای از آیتم‌ها از I است که با دو قانون زیر ساخته شده‌اند :

۱. هر آیتم در I در closure(I) نیز اضافه می‌شود.

۲. اگر $A \to \alpha \cdot B\beta$ در closure(I) باشد و $B \to \gamma$ یک قانون تولید باشد، آنگاه آیتم $B \to \cdot\gamma$ درصورتی که در closure(I) وجود نداشته باشد، به آن اضافه می‌شود. این کار تکرار می‌شود تا جایی که هیچ آیتم دیگری را نتوان به closure(I) اضافه کرد.

تابع Closure

- به طور شهودی $A \to \alpha \cdot B\beta$ در closure(I) نشان دهنده این است که در یکی از گام‌ها در فرایند تجزیه، زیررشته‌ای که باید تجزیه شود از $B\beta$ مشتق می‌شود. زیر رشته‌ای که از $B\beta$ مشتق یک پیشوند دارد که از B مشتق می‌شود بنابراین یکی از قوانین B باید اعمال شود. بنابراین آیتم‌ها برای همه قوانین متعلق به B را اضافه می‌شود. پس اگر $B \to \gamma$ یک قانون تولید باشد، $B \to \cdot\gamma$ در closure(I) قرار می‌گیرد.

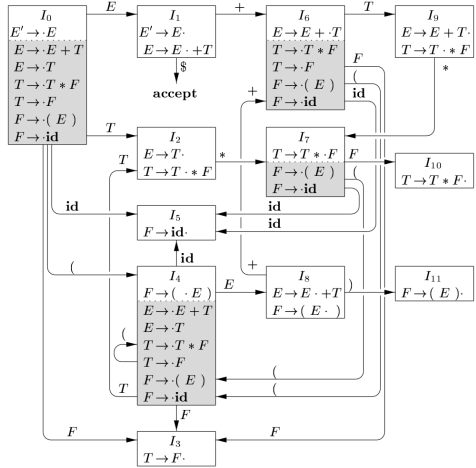
تابع Closure

- مثال : گرامر زیر را در نظر بگیرید.

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

تابع Closure

- اگر I مجموعه‌ای از یک آیتم $\{[E' \to \cdot E]\}$ باشد، آنگاه closure(I) مجموعه آیتم‌ها I_0 درشکل زیر را شامل می‌شود.



تابع Closure

- برای اینکه ببینیم تابع closure چگونه محاسبه شده است، $E' \rightarrow \cdot E$ در ابتدا براساس قانون اول در closure(I) قرار می‌گیرد.
- از آنجایی که E سمت راست نقطه قرارگرفته است، همه قوانین E را با یک نقطه در سمت چپ بدنه قانون می‌افزاییم : $E \rightarrow \cdot E + T$ و $E \rightarrow \cdot T$.
- در آیتم افزوده شده، T پس از نقطه قرارگرفته است پس قوانین $T \rightarrow \cdot T * F$ و $T \rightarrow \cdot F$ را می‌افزاییم.
- در پایان چون F پس از نقطه قرارگرفته است قوانین متعلق به F را با یک نقطه در سمت چپ بدنه قانون می‌افزاییم پس دو آیتم $F \rightarrow \cdot (E)$ و $F \rightarrow \cdot id$ اضافه می‌شوند.

تابع Closure

- تابع `closure` را می‌توان براساس الگوریتم زیر تولید کرد.

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production B → γ of G )
                if ( B → ·γ is not in J )
                    add B → ·γ to J;
    until no more items are added to J on one round;
    return J;
}
```

تابع Closure

- یک روش مناسب برای پیاده‌سازی `closure` نگهداری آرایه‌ای به نام `added` حاوی مقادیر منطقی است. مقدار `added[B]` به درست تغییر می‌کند هنگامی که آیتم‌های $B \rightarrow \cdot\gamma$ به ازای همه قوانین B افزوده می‌شوند.

تابع Closure

- توجه کنید اگر یک قانون متعلق به B به closure(I) با یک نقطه در سمت چپ افزوده شود، آنگاه همه قوانین B باید به closure اضافه شوند. بنابراین نیاز نیست همیشه همه آیتم‌های $B \rightarrow \cdot \gamma$ را که به closure(I) افزوده شده‌اند لیست کنیم. یک لیست از متغیرهای B که افزوده شده‌اند کافی است.

تابع Closure

- مجموعه‌های آیتم‌ها را به دو دسته تقسیم می‌کنیم.

۱. آیتم‌های هسته [1] : آیتم شروع $S' \rightarrow \cdot S$ و همه آیتم‌هایی که نقطه در سمت چپ بدنه آنها نیست.

۲. آیتم‌های غیرهسته [2] : همه آیتم‌هایی که نقطه در سمت چپ بدنه آنهاست به جز $S' \rightarrow \cdot S$.

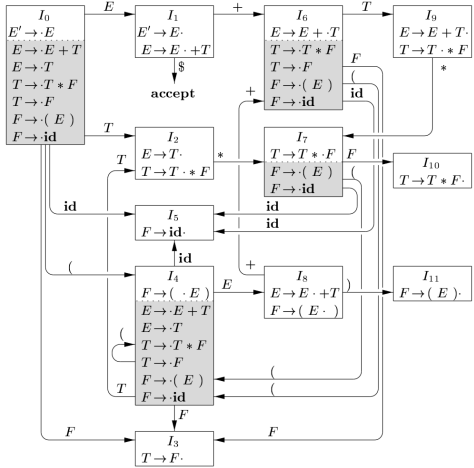
[1] kernel items
[2] Nonkernel items

تابع Closure

- هر مجموعه از آیتم‌ها تشکیل شده است از `closure` برروی مجموعه آیتم‌های هسته.
- بنابراین برای صرفه‌جویی در حافظه می‌توانیم آیتم‌های غیر هسته را دور بریزیم زیرا این آیتم‌ها مجددا می‌توانند از آیتم‌های هسته محاسبه شوند.

تابع Closure

- در شکل زیر آیتم‌های غیرهسته با رنگ خاکستری نشان داده شده‌اند.



تابع Closure

- یک تابع مهم و کاربردی دیگر $\texttt{Goto(I, X)}$ است. ورودی $\texttt{I}$ مجموعه‌ای از آیتم‌هاست و $\texttt{X}$ یک نماد از گرامر است.
- تابع $\texttt{Goto(I, X)}$ برابراست با $\texttt{closure}$ برروی مجموعه همهٔ آیتم‌های $[\texttt{A} \to \alpha\texttt{X} \cdot \beta]$ به طوری که $[\texttt{A} \to \alpha \cdot \texttt{X}\beta]$ در $\texttt{I}$ باشد.
- به طور شهودی، تابع $\texttt{Goto}$ برای تعریف گذارها در ماشین $\texttt{LR(0)}$ برای یک گرامر استفاده می‌شود. حالت‌های ماشین مجموعه‌ای از آیتم‌هاست و $\texttt{Goto(I, X)}$ گذار از حالت $\texttt{I}$ با ورودی $\texttt{X}$ است.

تابع Closure

- اگر I مجموعه‌ای از دو آیتم $\{[\mathrm{E}' \to \mathrm{E}\cdot], [\mathrm{E} \to \mathrm{E} \cdot +\mathrm{T}]\}$ باشد، آنگاه $\mathrm{Goto}(\mathrm{I}, +)$ شامل آیتم‌های زیر است.

$$
\begin{aligned}
E &\to E + \cdot T \\
T &\to \cdot T * F \\
T &\to \cdot F \\
F &\to \cdot(E) \\
F &\to \cdot \mathbf{id}
\end{aligned}
$$

تابع Closure

- برای محاسبه $\texttt{Goto(I, +)}$ همهٔ آیتم‌هایی را که در آنها $+$ پس از نقطه قرار می‌گیرند در نظر می‌گیریم. آیتم $\texttt{E} \to \texttt{E} \cdot +\texttt{T}$ یکی از این آیتم‌هاست. نقطه را به بعد از $+$ منتقل می‌کنیم و `closure` آن را محاسبه می‌کنیم.

تابع Closure

- الگوریتم زیر یک گروه استاندارد [1] از مجموعه‌های آیتم‌های LR(0) را برای گرامر افزوده شده G' می‌سازد.

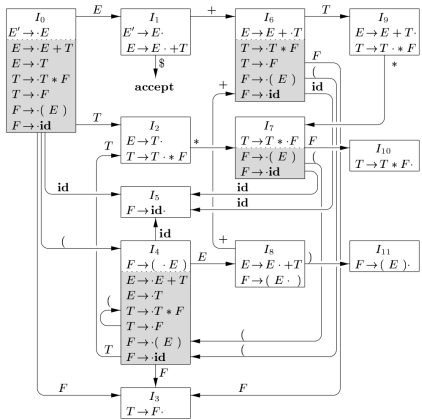
void $items(G')$ {
    $C = \{\text{CLOSURE}(\{[S' \to \cdot S]\})\}$;
    repeat
        for (each set of items I in C)
            for (each grammar symbol X)
                if ($\text{GOTO}(I, X)$ is not empty and not in C)
                    add $\text{GOTO}(I, X)$ to C;
    until no new sets of items are added to C on a round;
}

[1] canonical collection

تابع Closure

- گروه استاندارد از مجموعه‌های آیتم‌های LR(0) برای گرامر ذکر شده در شکل زیر نشان داده شده است.

$$\mathtt{E} \rightarrow \mathtt{E}+\mathtt{T} | \mathtt{T}\,,\ \mathtt{T} \rightarrow \mathtt{T}*\mathtt{T} | \mathtt{F} \rightarrow \mathtt{F} \rightarrow (\mathtt{E}) | \mathtt{id}$$



تابع Closure

- تابع Goto در واقع گذارها در شکل هستند.

استفاده از ماشین LR(0)

- تجزیه LR ساده یا SLR [1] از ماشین LR(0) استفاده می‌کند.
- حالت‌های ماشین LR(0) مجموعه‌هایی از آیتم‌های گروه‌های LR(0) استاندارد [2] است و گذارها با تابع Goto محاسبه شده‌اند.

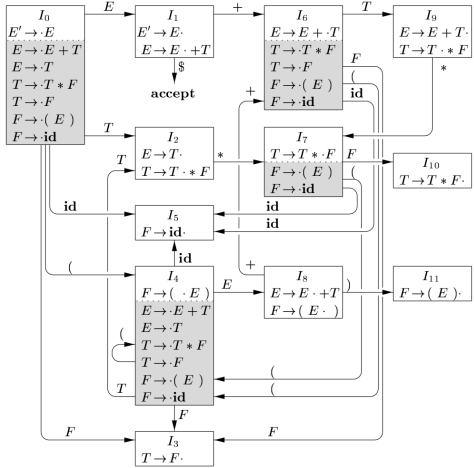
[1] simple LR
[2] sets of items from canonical LR(∘) collection

استفاده از ماشین LR(0)

- حالت آغازین ماشین LR(0) درواقع $\texttt{closure}([S' \to \cdot S])$ است، جایی که S' نماد آغازین گرامر افزوده شده است. همهٔ حالت‌ها حالت پذیرش هستند.
- وقتی می‌گوییم حالت j منظور مجموعه آیتم‌های I_j است.
- حال باید ببینیم ماشین LR(0) چگونه کمک می‌کند برای انتقال کاهش تصمیم بگیریم.
- فرض کنید رشته γ از نمادهای گرامر ماشین LR(0) را از حالت 0 به حالت j می‌برد. در اینصورت برروی نماد ورودی a انتقال انجام می‌دهیم اگر حالت j یک گذار با a دارد. در غیراینصورت یک کاهش انجام می‌دهیم که در اینصورت آیتم‌ها حالت j کمک می‌کنند تصمیم بگیریم از کدام قانون برای کاهش استفاده کنیم.
- الگوریتم تجزیه LR از یک پشته برای نگهداری حالت‌ها استفاده می‌کند.

استفاده از ماشین LR(0)

- رشته ورودی id * id و ماشین LR(0) زیر را در نظر بگیرید.



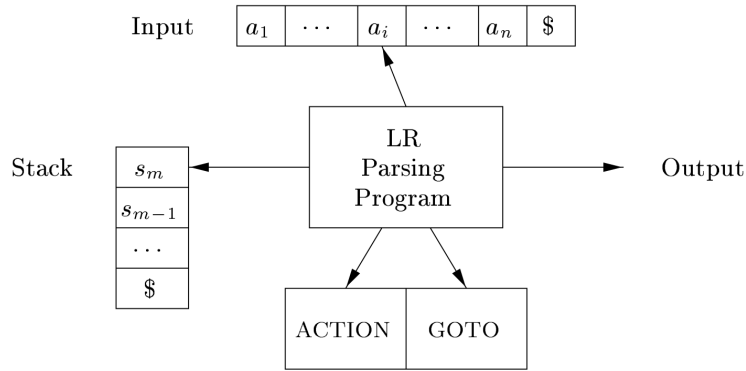
استفاده از ماشین LR(0)

- شکل زیر روند تجزیه $\texttt{id} * \texttt{id}$ را نشان می‌دهد.

Line	Stack	Symbols	Input	Action
(1)	0	\$	**id** $*$ **id** \$	shift to 5
(2)	0 5	\$ **id**	$*$ **id** \$	reduce by $F \to \mathbf{id}$
(3)	0 3	\$ F	$*$ **id** \$	reduce by $T \to F$
(4)	0 2	\$ T	$*$ **id** \$	shift to 7
(5)	0 2 7	\$ $T *$	**id** \$	shift to 5
(6)	0 2 7 5	\$ $T *$ **id**	\$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	\$ $T * F$	\$	reduce by $T \to T * F$
(8)	0 2	\$ T	\$	reduce by $E \to T$
(9)	0 1	\$ E	\$	accept

الگوریتم تجزیه LR

- شمای کلی یک تجزیه‌کننده LR در شکل زیر نمایش داده شده است.



الگوریتم تجزیه LR

- این تجزیه‌کننده از یک ورودی، یک خروجی، یک پشته، یک برنامه تجزیه‌کننده، و یک جدول تجزیه استفاده می‌کند که این جدول از دو بخش `Action` و `Goto` تشکیل شده است.
- برای تجزیه‌کننده برای همهٔ تجزیه‌کننده‌ها یکسان است. تنها جدول تجزیه به ازای هر تجزیه‌کننده متفاوت خواهد بود.
- برنامه تجزیه‌کننده کاراکترها را یک‌به‌یک از ورودی می‌خواند. جایی که یک تجزیه‌کننده انتقال کاهش یک انتقال انجام می‌دهد، تجزیه‌کنند LR یک حالت را انتقال می‌دهد. هرحالت درواقع اطلاعات حالت‌های پشته که در زیر آن قرار دارند را خلاصه می‌کند.
- پشته شامل حالت‌های $s_0 s_1 \cdots s_m$ می‌شود به طوری‌که s_m روی پشته است. پشته حالت‌های ماشین LR(0) را نگه می‌دارد.
- هر حالت یک نماد گرامر متناظر با آن دارد، زیر حالت‌ها متناظر هستند با مجموعه آیتم‌ها و یک گذار از حالت i به حالت j وجود دارد اگر $\texttt{Goto}(I_i, X) = I_j$
- همهٔ گذارها به حالت j برای نماد X هستند. بنابراین هرحالت به جز حالت 0 یک نماد متناظر با آن دارد.

ساختار جدول تجزیه LR

- جدول تجزیه از دو بخش تشکیل شده است : تابع Action و تابع Goto.

۱. تابع Action حالت i و یک ترمینال a (یا نماد $ که در پایان رشته است) را دریافت می‌کند. مقدار Action[i, a] یکی از چهار مورد زیر می‌تواند باشد.

- (a) انتقال حالت j : ورودی a به پشته منتقل می‌شود و حالت j نماینده ورودی a است.
- (b) کاهش $A \rightarrow \beta$: جمله β که برروی پشته قرار دارد با A کاهش پیدا می‌کند.
- (c) پذیرش : رشته پذیرفته می‌شود و تجزیه به اتمام می‌رسد.
- (d) خطا : تجزیه‌کننده یک خطا در ورودی می‌یابد و عملیات بازیابی خطا انجام می‌دهد.

۲. تابع Goto که برروی مجموعه‌های آیتم‌ها تعریف شده بود را به حالت‌ها تعمیم می‌دهیم. اگر $Goto[I_i, A] = I_j$ ، آنگاه Goto حالت i و متغیر A را به حالت j نگاشت می‌کند.

پیکربندی تجزیه کننده LR

- برای توصیف رفتار تجزیه‌کننده LR از یک نشانه‌گذاری برای نشان دادن وضعیت تجزیه‌کننده استفاده می‌کنیم، یعنی وضعیت پشته و رشته باقیمانده برای تجزیه.
- یک پیکربندی [1] از تجزیه کننده LR یک جفت به صورت $(s_0 s_1 \cdots s_m, a_i a_{i+1} \cdots a_n\$)$ است، به طوری‌که جزء اول محتوای پشته (بالای پشته در سمت راست) و جزء دوم باقیمانده رشته ورودی است.
- این پیکربندی نشان‌دهنده صورت جمله‌ای $X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$ است.
- درواقع X_i نماد گرامری است که با حالت s_i نمایش داده می‌شود.
- توجه کنید که حالت آغازین s_0 در تجزیه‌کننده نماینده هیچ نمادی از گرامر نیست و تنها زیر پشته را نشان می‌دهد.

[1] configuration

رفتار تجزیه کننده LR

- حرکت بعدی تجزیه‌کننده از یک پیکربندی با خواندن نماد ورودی a_i و حالت s_m برروی پشته توسط $\mathtt{Action}[s_m, a_i]$ از جدول تجزیه تعیین می‌شود.

۱. اگر $\mathtt{Action}[s_m, a_i] = \mathtt{shift}\ s$ باشد، آنگاه تجزیه‌کننده یک عملیات انتقال انجام می‌دهد و حالت بعدی s را به پشته منتقل می‌کند و در پیکربندی $(s_0 s_1 \cdots s_m s, a_{i+1} \cdots a_n \$)$ قرار می‌گیرد. نیازی نیست نماد a_i برروی پشته باشد زیرا اگر نیاز به آن بود می‌توان توسط حالت s آن را بازیابی کرد (البته در عمل هیچ‌گاه نیازی به آن نیست). نماد بعدی a_{i+1} خواهد بود.

۲. اگر $\mathtt{Action}[s_m, a_i] = \mathtt{reduce}\, A \to \beta$ باشد، آنگاه تجزیه‌کننده یک کاهش انجام می‌دهد و وارد پیکربندی $(s_0 s_1 \cdots s_{m-r} s, a_i a_{i+1} \cdots a_n \$)$ می‌شود. مقدار r طول β است و $s = \mathtt{Goto}[s_{m-r}, A]$. در اینجا تجزیه‌کننده ابتدا تعداد r حالت را از پشته برمی‌دارد و حالت s_{m-r} برروی پشته قرار می‌گیرد. سپس حالت s یعنی $\mathtt{Goto}[s_{m-r}, A]$ برروی پشته قرار می‌گیرد. برای تجزیه‌کننده‌های LR دنباله نمادهای $X_{m-r+1} \cdots X_m$ را می‌سازیم که متناظر است با حالت‌های برداشته شده از روی پشته که همیشه بر β یعنی بدنه قانون کاهش منطبق می‌شود.

رفتار تجزیه کننده LR

۳. اگر $\texttt{Action}[s_m, a_i] = \texttt{accept}$ آنگاه تجزیه به پایان می‌رسد.

۴. اگر $\texttt{Action}[s_m, a_i] = \texttt{error}$ آنگاه تجزیه‌کننده یک تابع بازیابی‌کننده خطا فراخوانی می‌کند.

الگوریتم تجزیه LR

- الگوریتم تجزیه‌کننده LR به صورت زیر عمل می‌کند. همهٔ تجزیه‌کننده‌های LR به همین صورت عمل می‌کنند و تنها تفاوت آنها اطلاعات ذخیره شده در Action و Goto در جدول تجزیه است.
- فرض کنید رشته w به یک تجزیه‌کننده LR برای گرامر G داده شده است.
- اگر رشته w متعلق به L(G) باشد، دنباله‌ای از قوانین کاهش به صورت پایین به بالا برای رشته w به دست می‌آید در غیراینصورت پیام خطا صادر می‌شود.

الگوریتم تجزیه LR

- در ابتدا تجزیه‌کننده s_0 را برروی پشته قرار می‌دهد و $w\$$ برروی بافر ورودی قرار می‌گیرد. سپس الگوریتم زیر اجرا می‌شود.

```
let a be the first symbol of w$;
while(1) {
    let s be the state on top of the stack;
    if ( ACTION[s, a] = shift t ) {
        push t onto the stack;
        let a be the next input symbol;
    } else if ( ACTION[s, a] = reduce A → β ) {
        pop |β| symbols off the stack;
        let state t now be on top of the stack;
        push GOTO[t, A] onto the stack;
        output the production A → β;
    } else if ( ACTION[s, a] = accept ) break;
    else call error-recovery routine;
}
```

الگوریتم تجزیه LR

- گرامر زیر را در نظر بگیرید.

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

الگوریتم تجزیه LR

- شکل زیر توابع `Action` و `Goto` از تجزیه کننده LR برای این گرامر نشان می‌دهد.

State	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

الگوریتم تجزیه LR

- در این جدول S_i به معنی انتقال و حالت پشته i است، r_j به معنی کاهش با قانون شماره j است، acc به معنی پذیرش و خانه‌های خالی به معنی خطا است.

الگوریتم تجزیه LR

- برای ورودی $\texttt{id} * \texttt{id} + \texttt{id}$ دنباله محتوای پشته و ورودی در شکل زیر نشان داده شده است.

	Stack	Symbols	Input	Action
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \to F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \to T * F$
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \to T$
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \to E + T$
(14)	0 1	E	$\$$	accept

الگوریتم تجزیه LR

- برای مثال در خط (۱) تجزیه‌کننده در حالت 0 است و اولین نماد `id` است. عملیات S5 باید انجام شود، بدین معنی که یک انتقال با وارد کردن حالت 5 به پشته انجام می‌شود. سپس * نماد بعدی است و عملیات حالت 5 بر روی ورودی * یک کاهش با $F \rightarrow \texttt{id}$ است. یک حالت از پشته برداشته می‌شود. چون `Goto` در حالت 0 برروی F حالت 3 است، پس حالت 3 برروی پشته اضافه می‌شود.

ساختن جدول تجزیه LR

- برای استفاده از تجزیه‌کننده LR ساده یا SLR ابتدا باید جدول تجزیه آن را بسازیم.
- الگوریتم SLR با آیتم‌های LR(0) و ماشین LR(0) آغاز می‌کند.
- به ازای گرامر دلخواه G گرامر افزوده شده G' با متغیر شروع جدید S' ساخته می‌شود. با استفاده از G' گروه استاندارد مجموعه آیتم‌های C با تابع Goto ساخته می‌شود.
- سپس جدول تجزیه با استفاده از جدول تجزیه زیر ساخته می‌شود. قبل از ساختن جدول نیاز داریم برای همهٔ متغیرهای A مقدار Follow(A) را محاسبه کنیم.

ساختن جدول تجزیه LR

- الگوریتم ساخت جدول تجزیه SLR به صورت زیر است :

۱. گروهی از مجموعه‌های آیتم‌های LR(0) برای گرامر G' به صورت $C = \{I_0, I_1, \cdots, I_n\}$ می‌سازیم.

۲. حالت i از I_i می‌سازیم و عملیات تجزیه برای حالت i را به صورت زیر تعیین می‌کنیم :

- (a) اگر $[A \to \alpha \cdot a\beta]$ در I_i باشد و $Goto(I_i, a) = I_j$ باشد، آنگاه $Action[i, a] = \texttt{shift j}$. در اینجا a باید یک ترمینال باشد.
- (b) اگر $[A \to \alpha\cdot]$ در I_i باشد، آنگاه $Action[i, a] = \texttt{reduce } A \to \alpha$ به ازای هر a در $Follow(A)$. در اینجا A نمی‌تواند S' باشد.
- (c) اگر $[S' \to S\cdot]$ در I_i باشد آنگاه $Action[i, \$] = \texttt{accep}$.

ساختن جدول تجزیه LR

- اگر در حین اجرای این الگوریتم تعارضی در عملیات به وجود آمد، می‌گوییم گرامر $\mathrm{SLR}(1)$ نیست و الگوریتم نمی‌تواند تجزیه‌کننده تولید کند.

۳. گذار $\mathtt{Goto}$ برای هر حالت $\mathtt{i}$ برای همه متغیرهای $\mathtt{A}$ با استفاده از این قانون محاسبه می‌شود : اگر $\mathtt{Goto}(\mathtt{I_i}, \mathtt{A}) = \mathtt{I_j}$ آنگاه $\mathtt{Goto}[\mathtt{i}, \mathtt{A}] = \mathtt{j}$.

۴. هر خانه‌ای در جدول که برای آن در گام‌ها ۲ و ۳ مقداری تولید نشده است، خطا محسوب می‌شود.

۵. حالت اولیه تجزیه‌کننده حالتی است که از مجموعه آیتم‌هایی ساخته شده است که حاوی $[\mathrm{S}' \to \cdot\mathrm{S}]$ است.

ساختن جدول تجزیه LR

- جدول تجزیه‌ای که با استفاده از این الگوریتم به دست می‌آید، جدول SLR(1) برای گرامر G نامیده می‌شود. تجزیه‌کننده LR که از جدول SLR(1) برای گرامر G استفاده می‌کند تجزیه‌کننده SLR(1) برای G نامیده می‌شود. گرامری که برای آن یک تجزیه‌کننده SLR(1) وجود داشته باشد، گرامر SLR(1) نامیده می‌شود. معمولا (۱) را حذف می‌کنیم و تجزیه‌کننده و گرامر را SLR می‌نامیم.

ساختن جدول تجزیه LR

- می‌خواهیم جدول تجزیه SLR برای گرامر زیر بسازیم.

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

- مجموعه آیتم‌های I_0 به صورت زیر است.

$$
\begin{aligned}
&E' \to \cdot E \\
&E \to \cdot E + T \\
&E \to \cdot T \\
&T \to \cdot T * F \\
&T \to \cdot F \\
&F \to \cdot (E) \\
&F \to \cdot \mathbf{id}
\end{aligned}
$$

ساختن جدول تجزیه LR

- با استفاده از آیتم $\mathtt{F} \to \cdot(\mathtt{E})$ می‌توان مقدار $\mathtt{Action}[0, (] = \mathtt{shift}\ 4$ را محاسبه کرد و با استفاده از آیتم $\mathtt{F} \to \cdot\mathtt{id}$ مقدار $\mathtt{Action}[0, \mathtt{id}] = \mathtt{shift}\ 5$ به دست می‌آید. بقیه آیتم‌ها در $\mathtt{I}_0$ مقداری به دست نمی‌دهند.
- حال آیتم‌های $\mathtt{I}_1$ را در نظر می‌گیریم. برای $\mathtt{E}' \to \mathtt{E}\cdot$ به دست می‌آوریم $\mathtt{Action}[1, \$] = \mathtt{accep}$ و برای $\mathtt{E} \to \mathtt{E} \cdot +\mathtt{T}$ به دست می‌آوریم $\mathtt{Action}[1, +] = \mathtt{shift}\ 6$.
- برای آیتم‌های $\mathtt{I}_2$ نیز عملیات را محاسبه می‌کنیم. برای آیتم $\mathtt{E} \to \mathtt{T}\cdot$ از آنجایی که $\mathtt{Follow}(\mathtt{E}) = \{\$, +,)\}$ داریم :

$$\mathtt{Action}[2, \$] = \mathtt{Action}[2, +] = \mathtt{Action}[2,)] = \mathtt{reduce}\ \ \mathtt{E} \to \mathtt{T}$$

- همچنین برای آیتم $\mathtt{T} \to \mathtt{T} \cdot *\mathtt{F}$ در $\mathtt{I}_2$ داریم $\mathtt{Action}[2, *] = \mathtt{shift}\ 7$.

ساختن جدول تجزیه LR

- با ادامه این روند جدول تجزیه به صورت زیر محاسبه می‌شود.

STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

ساختن جدول تجزیه LR

- هر گرامر SLR(1) غیرمبهم است، اما بسیاری از گرامرهای غیرمبهم وجود دارند که SLR(1) نیستند.
- گرامر زیر را در نظر بگیرید.

$$\begin{array}{lcl} S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathbf{id} \\ R & \rightarrow & L \end{array}$$

ساختن جدول تجزیه LR

- گروه استاندارد مجموعه‌های آیتم‌های LR(0) برای این گرامر به صورت زیر است.

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot = R$
$R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id}\cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L = R\cdot$

ساختن جدول تجزیه LR

- مجموعه آیتم‌های I_2 را در نظر بگیرید. با استفاده از آیتم اول به دست می‌آوریم $Action[2, =] = shift\ 6$. از آنجایی که $Follow(R)$ حاوی $=$ است (برای مثال در اشتقاق $S \Rightarrow L = R \Rightarrow R = R$)، آیتم دوم به دست می‌دهد $Action[2, =] = reduce\ R \rightarrow L$. چون برای $Action[2, =]$ دو مقدار انتقال و کاهش به دست می‌آوریم، پس در حالت ۲ برروی ورودی $=$ یک ناسازگاری انتقال و کاهش وجود دارد.
- این گرامر مبهم نیست. دلیل این ناسازگاری این است که تجزیه‌کننده SLR به اندازه کافی قدرتمند نیست تا بتواند برروی این گرامر تصمیم بگیرد. تجزیه‌کننده LALR مجموعه بزرگتری از گرامرها از جمله این گرامر را می‌تواند تجزیه کند.
- توجه‌کنید که گرامرهای غیرمبهمی وجود دارند که تجزیه‌کننده LR برای آنها وجود ندارد. البته این گرامرها در زبان‌های برنامه‌نویسی استفاده نمی‌شوند.

پیشوندهای ماندنی

- حال ببینیم چرا ماشین LR(0) می‌تواند برای تصمیم‌گیری درمورد انتقال و کاهش استفاده شود.
- در ماشین LR(0) برای یک گرامر، پشته پیشوندی از صورت جمله‌ای راست را نگهداری می‌کند.
- اگر پشته α را نگهداری کند و ورودی x باشد آنگاه دنباله‌ای از کاهش‌ها αx را به S کاهش می‌دهد. درواقع در فرایند اشتقاق داریم $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha x$.
- اما همه پیشوندهای صورت‌های جمله‌ای راست نمی‌توانند برروی پشته ظاهر شوند. برای مثال فرض کنید داشته‌باشیم $E * (E) \underset{rm}{\Rightarrow} F * id \underset{rm}{\overset{*}{\Rightarrow}} E$ آنگاه در فرایند تجزیه پشته می‌تواند حاوی) ، E ، (و (E) باشد اما نمی‌تواند حاوی *(E) باشد زیرا (E) یک هندل است که باید به F کاهش پیدا کند.

پیشوندهای ماندنی

- پیشوندهایی از صورت‌های جمله‌ای راست که می‌توانند برروی پشته دریک تجزیه‌کننده انتقال‌کاهش ظاهر شوند، پیشوندهای ماندنی [1] نامیده می‌شوند. یک پیشوند ماندنی پیشوندی از یک صورت جمله‌ای راست است که یک هندل را شامل نمی‌شود.
- تجزیه SLR بر این اصل عمل می‌کند که ماشین LR(0) پیشوندهای ماندنی را تشخیص می‌دهد. می‌گوییم آیتم $A \to \beta_1 \cdot \beta_2$ برای پیشوند ماندنی $\alpha\beta_1$ معتبر است اگر اشتقاقی به صورت $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$ وجود داشته باشد. یک آیتم می‌تواند برای بسیاری از پیشوندهای ماندنی معتبر باشد.
- این که $A \to \beta_1 \cdot \beta_2$ برای $\alpha\beta_1$ معتبر است به ما می‌گوید وقتی $\alpha\beta_1$ را برروی پشته مشاهده کردیم انتقال انجام دهیم یا کاهش.
- اگر $\beta_2 \neq \epsilon$ آنگاه می‌توان نتیجه گرفت که هندل به پشته انتقال داده نشده است بنابراین باید انتقال انجام شود. اگر $\beta_2 = \epsilon$ آنگاه $A \to \beta_1$ باید هندل باشد و باید با استفاده از این قانون کاهش انجام شود.

[1] viable prefix

پیشوندهای ماندنی

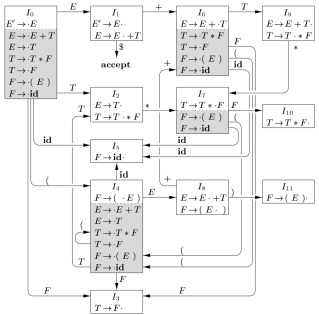
- البته اگر دو آیتم معتبر داشته باشیم که دو عملیات متفاوت را تعیین کنند، آنگاه ناسازگاری و تعارض به وجود می‌آید که در اینصورت بررسی کاراکترهای بعدی دروردی ممکن است به حل تعارض کمک کنند و درواقع تجزیه‌کننده‌های قدرتمندتر چنین می‌کنند.
- به سادگی می‌توانیم مجموعه آیتم‌های معتبر را برای هر پیشوند ماندنی که برروی پشته تجزیه‌کننده می‌تواند ظاهر شود را محاسبه کنیم.
- درواقع این یک قضیه پایه‌ای مهم در نظریه تجزیه LR است که مجموعه آیتم‌های معتبر برای پیشوند ماندنی γ دقیقا مجموعه آیتم‌هایی است که از حالت اولیه با مسیری با برچسب γ در ماشین LR(0) قابل دسترسی هستند.
- مجموعه همه آیتم‌های معتبر همه اطلاعات مهم که می‌توانند در پشته قرار بگیرند را در برمی‌گیرد. این قضیه در نظریه تجزیه‌کننده اثبات می‌شود که به اثبات آن نمی‌پردازیم.

پیشوندهای ماندنی

- گرامر زیر را در نظر بگیرید.

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

- مجموعهٔ آیتم‌های این گرامر به همراه توابع گذار را به صورت زیر قبلا محاسبه کردیم.



پیشوندهای ماندنی

- رشته $\mathtt{E}+\mathtt{T}*$ یک پیشوند ماندنی از گرامر است. ماشین LR(0) پس از خواندن $\mathtt{E}+\mathtt{T}*$ در حالت 7 قرار می‌گیرد. حالت 7 آیتم‌های زیر را شامل می‌شود که برای $\mathtt{E}+\mathtt{T}*$ معتبر هستند.

$$T \to T * \cdot F$$
$$F \to \cdot(E)$$
$$F \to \cdot\mathbf{id}$$

پیشوندهای ماندنی

- برای درک دلیل این امر اشتقاق راست زیر را در نظر بگیرید.

$$
\begin{aligned}
E' &\underset{rm}{\Rightarrow} E \\
&\underset{rm}{\Rightarrow} E + T \\
&\underset{rm}{\Rightarrow} E + T * F
\end{aligned}
\qquad
\begin{aligned}
E' &\underset{rm}{\Rightarrow} E \\
&\underset{rm}{\Rightarrow} E + T \\
&\underset{rm}{\Rightarrow} E + T * F \\
&\underset{rm}{\Rightarrow} E + T * (E)
\end{aligned}
\qquad
\begin{aligned}
E' &\underset{rm}{\Rightarrow} E \\
&\underset{rm}{\Rightarrow} E + T \\
&\underset{rm}{\Rightarrow} E + T * F \\
&\underset{rm}{\Rightarrow} E + T * \mathbf{id}
\end{aligned}
$$

پیشوندهای ماندنی

- اشتقاق اول معتبر بودن $T \to T * \cdot F$ را نشان می‌دهد. اشتقاق دوم معتبر بودن $F \to \cdot (E)$ و اشتقاق سوم معتبر بود $F \to \cdot \texttt{id}$ را نشان می‌دهد.
- می‌توان نشان داد که هیچ آیتم معتبری برای $E + T*$ وجود ندارد.

تجزیه‌کننده‌های LR قدرتمندتر

- در این قسمت تجزیه‌کننده‌های LR را تعمیم می‌دهیم و دو تجزیه‌کننده قدرتمند را شرح می‌دهیم.

۱. تجزیه‌کننده LR استاندارد [1] یا CLR : این روش از مجموعه‌ای بزرگ از آیتم‌ها به نام آیتم‌های LR(1) استفاده می‌کند.

۲. تجزیه‌کننده LR با بررسی نماد جلویی [2] یا LALR : این روش بر پایه مجموعه‌های آیتم‌های LR(0) است و تعداد بسیار کمتری حالت نسبت به تجزیه‌کننده‌ها بر پایه LR(1) دارد. تجزیه‌کننده LALR تعداد بسیار بیشتری از گرامرها را نسبت به SLR پوشش می‌دهد و جدول تجزیه‌ای که از آن استفاده می‌کند از جدول‌های SLR بزرگ‌تر نیست. در بسیاری از تجزیه‌کننده‌ها و کامپایلرها از روش LALR استفاده می‌شود.

[1] canonical LR
[2] lookahead LR

آیتم‌های LR(1) استاندارد

- می‌خواهیم یکی از روش‌های بسیار متداول برای تولید جدول تجزیه LR را شرح دهیم.
- در روش SLR ، حالت $\mathtt{i}$ عملیات کاهش را با $\mathtt{A} \to \alpha$ انجام می‌دهد اگر مجموعه آیتم‌های $\mathtt{I_i}$ شامل $[\mathtt{A} \to \alpha\cdot]$ باشد و نماد ورودی $\mathtt{a}$ در $\mathtt{Follow(A)}$ باشد.
- در برخی مواقع وقتی حالت $\mathtt{i}$ برروی پشته است، پیشوند ماندنی $\beta\alpha$ برروی پشته چنان است که $\beta\mathtt{A}$ نمی‌تواند با $\mathtt{a}$ در هیچ یک از صورت‌های جمله‌ای دنبال شود. در چنین مواردی کاهش $\mathtt{A} \to \alpha$ برروی ورودی $\mathtt{a}$ غیر معتبر است.

آیتم‌های LR(1) استاندارد

- مثال : گرامر زیر را در نظر بگیرید.

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

آیتم‌های LR(1) استاندارد

- مجموعه آیتم‌های LR(0) را برای این گرامر به صورت زیر ساختیم.

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S\cdot$

I_2: $S \to L\cdot = R$
$R \to L\cdot$

I_3: $S \to R\cdot$

I_4: $L \to *\cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id}\cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to *R\cdot$

I_8: $R \to L\cdot$

I_9: $S \to L = R\cdot$

آیتم‌های LR(1) استاندارد

- در حالت ۲ آیتم $R \to L\cdot$ را داشتیم. فرض کنید این قانون همان $A \to \alpha$ است و نماد a در ورودی در اینجا علامت $=$ است که در $\texttt{Follow}(R)$ است.
- تجزیه‌کننده SLR کاهش با استفاده از $R \to L$ را در حالت ۲ با خواندن نماد $=$ فراخوانی می‌کند.
- اما هیچ صورت جمله‌ای در این گرامر که با $R = \ldots$ آغاز شود وجود ندارد. بنابراین در حالت ۲ که حالت متناظر با پیشوند ماندنی L است نباید L را با R کاهش دهد.

آیتم‌های LR(1) استاندارد

- در چنین مواردی نیاز داریم اطلاعات بیشتری دریافت کنیم تا به ما کمک کند برخی از کاهش‌ها را انجام ندهیم.
- می‌توانیم در هریک از حالت‌ها تجزیه‌کننده LR مشخص کنیم کدام نمادها می‌توانند هندل α را دنبال کنند وقتی یک کاهش به صورت $A \rightarrow \alpha$ وجود داشته باشد.
- این اطلاعات اضافی را بدین صورت در جدول تجزیه درج می‌کنیم که آیتم‌ها یک ترمینال را به عنوان مؤلفه دوم شامل شوند.
- بنابراین یک آیتم به صورت $[A \rightarrow \alpha \cdot \beta \ , \ a]$ درمی‌آید که در آن $A \rightarrow \alpha\beta$ یک قانون تولید و a یک نماد الفبا یا نماد پایان رشته \$ است. چنین آیتم‌هایی را آیتم LR(1) می‌نامیم.

آیتم‌های LR(1) استاندارد

- عدد ۱ در LR(1) طول مؤلفه دوم در آیتم است که نمادهای جلویی [1] در آیتم نامیده می‌شود.
- نماد جلویی هیچ تأثیری در آیتمی که به صورت $[A \to \alpha \cdot \beta \ , \ a]$ است ندارد اگر β رشته تهی نباشد. اما در آیتم $[A \to \alpha \cdot \ , \ a]$ کاهش با استفاده از $A \to \alpha$ تنها صورتی انجام می‌شود که نماد بعدی a باشد.
- می‌گوییم آیتم LR(1) به صورت $[A\alpha \cdot \beta \ , \ a]$ برای پیشوند ماندنی γ معتبر است اگر اشتقاقی به صورت $S \underset{rm}{\overset{*}{\Rightarrow}} \delta Aw \underset{rm}{\Rightarrow} \delta\alpha\beta w$ وجود داشته باشد که (۱) $\gamma = \delta\alpha$ باشد و (۲) یا a اولین نماد w باشد و یا w تهی باشد و a نماد $ باشد.

[1] lookahead

آیتم‌های LR(1) استاندارد

- مثال : گرامر زیر را در نظر بگیرید.

$$S \to B\ B$$
$$B \to a\ B \mid b$$

- یک اشتقاق راست $\mathtt{aaaBab} \underset{rm}{\Rightarrow} \mathtt{aaBab} \overset{*}{\underset{rm}{\Rightarrow}} \mathtt{S}$ وجود دارد. می‌بینیم که آیتم $[\mathtt{B} \to \mathtt{a} \cdot \mathtt{B}, \mathtt{a}]$ برای پیشوند ماندنی $\gamma = \mathtt{aaa}$ معتبر است اگر $\delta = \mathtt{aa}$ و $\mathtt{A} = \mathtt{B}$ و $\mathtt{w} = \mathtt{ab}$ و $\alpha = \mathtt{a}$ و $\beta = \mathtt{B}$ باشد.
همچنین اشتقاق $\mathtt{BaaB} \underset{rm}{\Rightarrow} \mathtt{BaB} \overset{*}{\underset{rm}{\Rightarrow}} \mathtt{S}$ وجود دارد. از این اشتقاق می‌بینیم که آیتم $[\mathtt{B} \to \mathtt{a} \cdot \mathtt{B}, \$]$ برای پیشوند ماندنی Baa معتبر است.

ساختن مجموعه‌های آیتم‌های LR(1)

- روش ساختن گروه مجموعه‌های آیتم‌های LR(1) معتبر شبیه ساخت مجموعه آیتم‌های LR(0) است. تنها تفاوت در توابع `Closure` و `Goto` است.

ساختن مجموعه‌های آیتم‌های LR(1)

- برای گرامر G' مجموعه آیتم‌های LR(1) با استفاده از الگوریتم زیر محاسبه می‌شود.

```
SetOfItems CLOSURE(I) {
    repeat
        for ( each item [A → α·Bβ, a] in I )
            for ( each production B → γ in G' )
                for ( each terminal b in FIRST(βa) )
                    add [B → ·γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for ( each item [A → α·Xβ, a] in I )
        add item [A → αX·β, a] to set J;
    return CLOSURE(J);
}

void items(G') {
    initialize C to {CLOSURE({[S' → ·S, $]})};
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C;
}
```

ساختن مجموعه‌های آیتم‌های LR(1)

- برای اینکه بفهمیم چرا b باید در $\texttt{First}(\beta a)$ باشد، آیتمی به صورت $[A \to \alpha \cdot B\beta, a]$ را در نظر بگیرید که در مجموعه آیتم‌های معتبر برای پیشوند ماندنی γ است. آنگاه اشتقاق راست $S \underset{rm}{\overset{*}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta\alpha B\beta ax$ وجود دارد به طوری که $\gamma = \delta\alpha$.
- فرض کنید βax رشته by را مشتق کند. آنگاه برای هر قانون $B \to \eta$ اشتقاق $S \underset{rm}{\overset{*}{\Rightarrow}} \gamma Bby \underset{rm}{\Rightarrow} \gamma\eta by$ را داریم. بنابراین $[B \to \cdot\eta, b]$ برای γ معتبر است.
- توجه کنید که b می‌تواند اولین ترمینال مشتق شده از β باشد یا ممکن است β رشته تهی در اشتقاق $\beta ax \underset{rm}{\overset{*}{\Rightarrow}} by$ مشتق کند.
- برای خلاصه این دو احتمال، می‌گوییم b می‌تواند هر ترمینالی در $\texttt{First}(\beta ax)$ باشد. توجه کنید که x نمی‌تواند اولین ترمینال by را شامل شود پس $\texttt{First}(\beta ax) = \texttt{First}(\beta a)$.

ساختن مجموعه‌های آیتم‌های LR(1)

- مثال : گرامر زیر را در نظر بگیرید.

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & C\ C \\ C & \rightarrow & c\ C \mid d \end{array}$$

- ابتدا closur برروی $[S' \rightarrow \cdot S, \$]$ را محاسبه می‌کنیم. آیتم $[S' \rightarrow \cdot S, \$]$ را بر $[A \rightarrow \alpha \cdot B\beta, a]$ تطبیق می‌دهیم. پس $A = S'$ ، $\alpha = \epsilon$ ، $B = S$ ، $\beta = \beta$ و $a = \$$ است. تابع Closure می‌گوید برای هرقانون $B \rightarrow \gamma$ آیتم $[B \rightarrow \cdot\gamma, b]$ را به ازای هر b در $First(\beta a)$ اضافه کنیم. در این گرامر $B \rightarrow \gamma$ باید $S \rightarrow CC$ باشد زیرا β تهی است و a نماد $ است، پس b تنها می‌تواند $ باشد. پس آیتم $[S \rightarrow \cdot CC, \$]$ را می‌افزاییم.

ساختن مجموعه‌های آیتم‌های LR(1)

- برای ادامه محاسبه closure همهٔ آیتم‌های $[\mathtt{C} \to \cdot\gamma, \mathtt{b}]$ را برای b در $\mathtt{First(C\$)}$ اضافه می‌کنیم. با تطبیق $[\mathtt{S} \to \cdot\mathtt{CC}, \$]$ بر $[\mathtt{A} \to \alpha \cdot \mathtt{B}\beta, \mathtt{a}]$ خواهیم داشت $\mathtt{A} = \mathtt{S}$ ، $\alpha = \epsilon$ ، $\mathtt{B} = \mathtt{C}$ و $\beta = \mathtt{C}$. از آنجایی که $\mathtt{a} = \$$. از آنجایی که C رشته تهی تولید نمی‌کند، $\mathtt{First(C\$)} = \mathtt{First(C)}$ ترمینال‌های c و d را شامل می‌شود، آیتم‌های $[\mathtt{C} \to \cdot\mathtt{cC}, \mathtt{c}]$ و $[\mathtt{C} \to \cdot\mathtt{cC}, \mathtt{d}]$ و $[\mathtt{C} \to \cdot\mathtt{d}, \mathtt{c}]$ و $[\mathtt{C} \to \cdot\mathtt{d}, \mathtt{d}]$ را می‌افزاییم. هیچ‌کدام از این آیتم‌ها در سمت راست نقطه متغیر ندارد، بنابراین محاسبه اولین مجموعه LR(0) به اتمام می‌رسد.

$$
\begin{aligned}
I_0: \quad & S \to \cdot S, \$ \\
& S \to \cdot CC, \ \$ \\
& C \to \cdot cC, \ c/d \\
& C \to \cdot d, \ c/d
\end{aligned}
$$

- در اینجا برای سادگی به جای دو آیتم $[\mathtt{C} \to \cdot\mathtt{cC}, \mathtt{c}]$ و $[\mathtt{C} \to \cdot\mathtt{cC}, \mathtt{d}]$ می‌نویسیم $\mathtt{c/d}$ و $\mathtt{C} \to \cdot\mathtt{cC}$.

ساختن مجموعه‌های آیتم‌های LR(1)

- حال باید $\texttt{Goto}(\texttt{I}_0, \texttt{X})$ را برای مقادیر مختلف X محاسبه کنیم. اگر $\texttt{X} = \texttt{S}$ باشد آیتم $[\texttt{S}' \to \texttt{S}\cdot, \$]$ به دست می‌آید. بنابراین داریم.

$$I_1 : \quad S' \to S\cdot,\ \$$$

- برای $\texttt{X} = \texttt{C}$ آیتم $[\texttt{S} \to \texttt{C} \cdot \texttt{C}, \$]$ به وجود می‌آید. همهٔ قوانین متغیر C را با نماد $ به عنوان مؤلفه دوم می‌افزاییم و به دست می‌آوریم :

$$\begin{aligned} I_2 : \quad & S \to C\cdot C,\ \$ \\ & C \to \cdot cC,\ \$ \\ & C \to \cdot d,\ \$ \end{aligned}$$

ساختن مجموعه‌های آیتم‌های LR(1)

- اگر داشته باشیم $\texttt{X} = \texttt{c}$ آنگاه آیتم $[\texttt{C} \to \texttt{c} \cdot \texttt{C}, \texttt{c/d}]$ را به دست می‌آوریم. همه قوانین متغیر C را با نماد c/d به عنوان مؤلفه دوم می‌افزاییم.

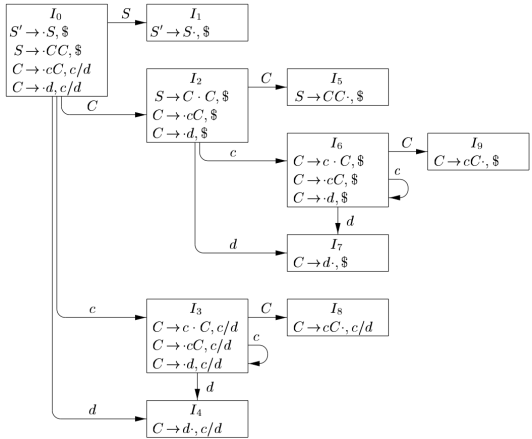
$$
\begin{aligned}
I_3 : \quad & C \to c \cdot C,\ c/d \\
& C \to \cdot cC,\ c/d \\
& C \to \cdot d,\ c/d
\end{aligned}
$$

- در نهایت قرار می‌دهیم $\texttt{X} = \texttt{d}$ و به دست می‌آوریم.

$$
I_4 : \quad C \to d \cdot,\ c/d
$$

ساختن مجموعه‌های آیتم‌های LR(1)

- به همین ترتیب با محاسبه آیتم‌ها و توابع Goto گراف زیر را به دست می‌آوریم.



جداول تجزیه LR(1) استاندارد

- حال الگوریتمی را توصیف می‌کنیم که برای یک گرامر جدول تجزیه LR استاندارد با توابع Action و Goto تولید می‌کند.

۱. برای گرامر افزوده شده G' گروه مجموعه‌های آیتم‌های LR(1) را به صورت $C' = \{I_0, I_1, \cdots, I_n\}$ می‌سازیم.

۲. حالت i از تجزیه‌کننده از I_i ساخته می‌شود. عملیات تجزیه برای i به صورت زیر تعیین می‌شود.

- (a) اگر $[A \to \alpha \cdot a\beta, b]$ در I_i باشد و $Goto(I_i, a) = I_j$ باشد، آنگاه $Action[i, a] = shift\ j$. در اینجا a باید یک ترمینال باشد.
- (b) اگر $[A \to \alpha\cdot, a]$ در I_i باشد و $A \neq S'$ ، آنگاه $Action[i, a] = reduce\ \ A \to \alpha$.
- (c) اگر $[S' \to S\cdot, \$]$ در I_i باشد، آنگاه $Action[i, \$] = accept$ اگر هر ناسازگاری در عملیات بالا رخ دهد می‌گوییم گرامر LR(1) نیست.

جداول تجزیه LR(1) استاندارد

۳. توابع گذار Goto برای حالت i برای همهٔ متغیرهای A به صورت زیر ساخته می‌شود : اگر $Goto(I_i, A) = I_j$ ، آنگاه $Goto[i, A] = j$.

۴. همهٔ خانه‌هایی که در گام‌های (۲) و (۳) تعریف نشده‌اند، خطا محسوب می‌شوند.

۵. حالت اولیه تجزیه‌کننده، حالتی است که از مجموعه آیتم‌های حاوی $[S' \to \cdot S, \$]$ تشکیل شده است.

جداول تجزیه LR(1) استاندارد

- جدولی که با استفاده از الگوریتم قبل تولید می‌شود جدول تجزیه LR(1) استاندارد [1] نامیده می‌شود.
- یک تجزیه‌کنندهٔ LR که از چنین جدولی استفاده کند، تجزیه‌کننده LR(1) استاندارد نامیده می‌شود.
- گرامری که برای آن یک تجزیه LR(1) استاندارد وجود داشته باشد گرامر LR(1) نامیده می‌شود.

[1] canonical LR(۱)

جداول تجزیه LR(1) استاندارد

- گرامر زیر را در نظر بگیرید.

$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & C\ C \\
C & \rightarrow & c\ C \mid d
\end{array}
$$

جداول تجزیه LR(1) استاندارد

- جدول تجزیه LR(1) استاندارد برای این گرامر در زیر نشان داده شده است. قوانین ۱، ۲ و ۳ به ترتیب $S \to CC$ ، $C \to cC$ و $C \to d$ هستند.

State	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

جداول تجزیه LR(1) استاندارد

- هر گرامر SLR(1) یک گرامر LR(1) است اما برای یک گرامر SLR(1) تجزیه کننده LR استاندارد ممکن است تعداد حالت‌های بیشتری نسبت به تجزیه‌کننده SLR برای همان گرامر داشته باشد.

ساختن جداول تجزیه LALR

- حال به معرفی تجزیه‌کننده LALR [1] می‌پردازیم. این تجزیه‌کننده در عمل بسیار مورد استفاده قرار می‌گیرد، زیرا جداول تجزیه آن از جدول تجزیه‌کننده LR استاندارد کوچک‌تر هستند و بیشتر زبان‌های برنامه‌نویسی را می‌توان با استفاده از گرامر LALR می‌توان توصیف کرد.
- جداول SLR نیز نسبت به CLR کوچک‌تر هستند اما برخی از ساختارهای زبان‌های برنامه‌نویسی را نمی‌توان با استفاده از گرامرهای SLR توصیف کرد.
- جداول تجزیه SLR و LALR تقریبا برابرند و در زبانی مانند زبان سی حدود چند صد حالت دارند، اما جدول تجزیه CLR برای زبان سی حدود چند هزار حالت خواهد داشت.

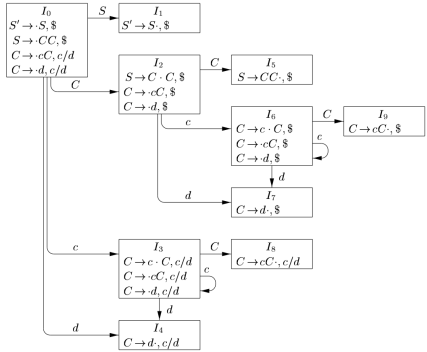
[1] lookahead LR

ساختن جداول تجزیه LALR

- گرامر زیر را در نظر بگیرید.

$$
\begin{array}{rcl}
S' & \rightarrow & S \\
S & \rightarrow & C\ C \\
C & \rightarrow & c\ C \mid d
\end{array}
$$

- مجموعه آیتم‌های LR(1) برای این گرامر در شکل زیر نشان داده شده‌اند.



ساختن جداول تجزیه LALR

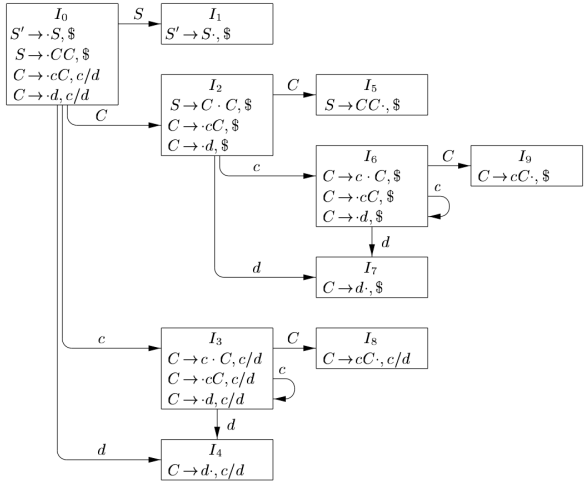
- دو حالت شبیه به هم I_4 و I_7 را در نظر بگیرید. هریک از این حالت‌ها فقط آیتم‌هایی با مؤلفه اول $C \to d\cdot$ دارند. در I_4 مؤلفه دوم c یا d است و در I_7 مؤلفه دوم $ است.
- برای اینکه تفاوت I_4 و I_7 را بفهمیم، یک مثال را بررسی می‌کنیم. این گرامر زبان منظم c^*dc^*d را تولید می‌کند. وقتی ورودی $cc\cdots cdcc\cdots cd$ خوانده می‌شود، تجزیه‌کننده اولین گروه از c ها و کاراکتر d پس از آن را به پشته انتقال می‌دهد و وارد حالت ۴ می‌شود. سپس کاهش توسط $C \to d$ انجام می‌شود با توجه به اینکه کاراکتر بعدی می‌تواند c یا d باشد. اگر $ به دنبال اولین d بیاید، یک ورودی به صورت ccd داریم که در زبان نیست و حالت ۴ به درستی با خواندن $ وجود خطا را نشان می‌دهد.

ساختن جداول تجزیه LALR

- تجزیه‌کننده بعد از خواندن دومین d وارد حالت ۷ می‌شود. پس از آن باید در ورودی $ خوانده شود وگرنه ورودی برطبق الگوی s^*dc^*d نیست. بنابراین حالت ۷ و ورودی $ کاهش $C \to d$ انجام می‌شود و با ورودی c یا d خطا صادر می‌شود.
- حال فرض‌کنید I_4 و I_7 را با I_{47} جایگزین کنیم که اجتماع I_4 و I_7 است و از سه آیتم $[C \to d\cdot, c/d/\$]$ تشکیل شده است.
- عملیات حالت 47 اکنون این است که بر روی کاهش انجام می‌دهد در حالی که قبل از ادغام دو حالت برخی از شرایط منجر به خطا می‌شدند. البته خطا اکنون نیز تشخیص داده خواهد شد.
- در حالت کلی می‌توانیم آیتم‌هایی را که هسته یکسان دارند یا به عبارت دیگر مؤلفه اول آنها یکسان است ادغام کنیم.

ساختن جداول تجزیه LALR

- آیتم‌های زیر را در نظر بگیرید.



ساختن جداول تجزیه LALR

- برای مثال در I_4 و I_7 آیتم‌هایی با هسته $\{C \to d\cdot\}$ وجود دارد. همچنین در I_3 و I_6 هسته $\{C \to c \cdot C, C \to \cdot cC, C \to \cdot d\}$ وجود دارد. در I_8 و I_9 نیز هسته $C \to cC\cdot$ وجود دارد.
- یک هسته یک مجموعه از آیتم‌های LR(0) برای یک گرامر است و یک گرامر LR(1) ممکن است بیش از دو مجموعه از آیتم‌ها با یک هسته تولید کند.

ساختن جداول تجزیه LALR

- از آنجایی که هسته $\texttt{Goto}(\texttt{I}, \texttt{X})$ فقط به هسته $\texttt{I}$ بستگی دارد، توابع $\texttt{Goto}$ از مجموعه‌های ادغام شده نیز ادغام می‌شوند.
- فرض‌کنید یک گرامر $\texttt{LR}(1)$ داریم. اگر همهٔ حالت‌ها با هستهٔ یکسان را ادغام کنیم، این احتمال وجود دارد که نتیجه دارای ناسازگاری باشد اما به احتمال زیاد ناسازگاری رخ نخواهد داد.
- می‌توان اثبات کرد که حاصل ادغام هیچ‌گاه ناسازگاری انتقال‌کاهش نخواهد داشت، اما این امکان وجود دارد که ناسازگاری کاهش‌کاهش رخ دهد.

ساختن جداول تجزیه LALR

- مثال : گرامر زیر را در نظر بگیرید.

$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e \\
A & \rightarrow & c \\
B & \rightarrow & c
\end{array}
$$

- این گرامر چهار رشته acd و ace و bcd و bce را تولید می‌کند. این گرامر یک گرامر LR(1) است.
- مجموعه آیتم‌های $\{[\mathtt{A} \rightarrow \mathtt{c}\cdot, \mathtt{d}], [\mathtt{B} \rightarrow \mathtt{c}\cdot, \mathtt{e}]\}$ برای پیشوند ماندنی ac معتبر است و $\{[\mathtt{A} \rightarrow \mathtt{c}\cdot, \mathtt{e}], [\mathtt{B} \rightarrow \mathtt{c}\cdot, \mathtt{d}]\}$ برای bc معتبر است. هیچ‌کدام از این مجموعه‌ها ناسازگاری ندارند. اما اجتماع آنها ناسازگاری کاهش‌کاهش ایجاد می‌کند.

$$
\begin{array}{l}
A \rightarrow c\cdot,\ d/e \\
B \rightarrow c\cdot,\ d/e
\end{array}
$$

ساختن جداول تجزیه LALR

- برای ساخت جدول LALR ابتدا مجموعه آیتم‌های LR(1) را می‌سازیم و اگر ناسازگاری ایجاد نشود هسته‌های یکسان را ادغام می‌کنیم. سپس یک جدول تجزیه از آیتم‌های ادغام شده می‌سازیم. اگر امکاه چنین جدولی وجود داشت گرامر LALR(1) است.

ساختن جداول تجزیه LALR

- یک الگوریتم ساده برای ساخت جدول تجزیه LALR به شرح زیر است.

۱. گروه مجموعه آیتم‌های $LR(1)$ را به صورت $C' = \{I_0, I_1, \cdots, I_n\}$ می‌سازیم.

۲. برای هر هسته در بین مجموعه آیتم‌های $LR(1)$ مجموعه‌هایی هسته یکسان دارند را پیدا کرده مجموعه‌های آنها را ادغام می‌کنیم.

۳. فرض‌کنیم مجموعه‌های آیتم‌های $LR(1)$ به دست آمده $C' = \{J_0, J_1, \cdots, J_m\}$ است. عملیات برای حالت i از آیتم J_i به همان صورتی که قبلا توضیح داده شده به دست می‌آید. اگر یک ناسازگاری وجود داشته باشد، تجزیه‌کننده $LALR(1)$ نیست.

۴. تابع Goto به صورت زیر محاسبه می‌شود. اگر J اجتماع یک یا مجموعه آیتم‌های $LR(1)$ باشد یعنی $J = I_1 \cup I_2 \cup \cdots \cup I_k$ آنگاه هسته‌ها $Goto(I_1, X)$ و $Goto(I_2, X)$ و $\cdots$ و $Goto(I_k, X)$ یکسان هستند زیر $I_1, I_2, \cdots, I_k$ هسته یکسان دارند. فرض‌کنید k اجتماع همه مجموعه‌های آیتم‌هایی باشد که هسته آنها $Goto(I_1, X)$ است. آنگاه $Goto(J, X)$.

ساختن جداول تجزیه LALR

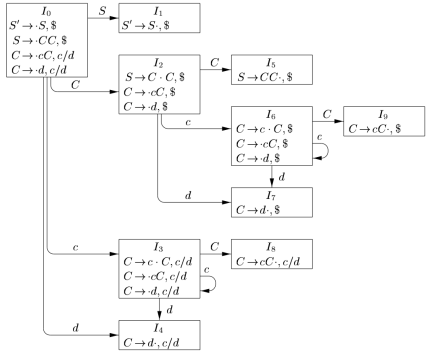
- جدولی که از الگوریتم قبل به دست می‌آید جدول تجزیه LALR نامیده می‌شود و اگر ناسازگاری وجود نداشته باشد گرامر به دست آمده LALR(1) نامیده می‌شود.

ساختن جداول تجزیه LALR

- مثال : گرامر زیر را در نظر بگیرید.

$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & C\ C \\
C & \rightarrow & c\ C \mid d
\end{array}
$$

- گراف توابع Goto برای این گرامر در زیر نشان داده شده است.



ساختن جداول تجزیه LALR

- در مجموعه آیتم‌ها می‌توانیم سه جفت از آیتم‌ها را ادغام کنیم.
- مجموعه آیتم‌های $\mathtt{I}_3$ و $\mathtt{I}_6$ به صورت زیر می‌توانند ادغام شوند.

$$\begin{array}{ll} I_{36}: & C \to c{\cdot}C,\ c/d/\$ \\ & C \to {\cdot}cC,\ c/d/\$ \\ & C \to {\cdot}d,\ c/d/\$ \end{array}$$

- مجموعه آیتم‌های $\mathtt{I}_4$ و $\mathtt{I}_7$ را می‌توانیم به صورت زیر ادغام کنیم.

$$I_{47}: \quad C \to d{\cdot},\ c/d/\$$$

- و همچنین مجموعه آیتم‌های $\mathtt{I}_8$ و $\mathtt{I}_9$ را می‌توانیم ادغام کنیم.

$$I_{89}: \quad C \to cC{\cdot},\ c/d/\$$$

ساختن جداول تجزیه LALR

- جدول LALR به صورت زیر به دست خواهد آمد.

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

- تابع $\texttt{Goto}(\texttt{I}_{36}, \texttt{C})$ را در نظر بگیرید. در مجموعه آیتم‌های $\texttt{LR(1)}$ داریم $\texttt{Goto}(\texttt{I}_3, \texttt{C}) = \texttt{I}_8$ و $\texttt{I}_8$ اکنون عضوی از $\texttt{I}_{89}$ است، بنابراین $\texttt{Goto}(\texttt{I}_{36}, \texttt{C}) = \texttt{I}_{89}$. از طرف دیگر اگر $\texttt{I}_6$ را در نظر بگیریم به همین نتیجه می‌رسیم، زیرا $\texttt{Goto}(\texttt{I}_6, \texttt{C}) = \texttt{I}_9$ و $\texttt{I}_9$ نیز عضوی از $\texttt{I}_{89}$ است.

ساختن جداول تجزیه LALR

- زبان c^*dc^*d را بار دیگر در نظر بگیرید. تجزیه‌کننده LR و تجزیه‌کننده LALR برای این زبان شبیه به یکدیگر عمل می‌کنند و دنباله عملیات انتقال و کاهش مشابه انجام می‌دهند.

State	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

State	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

ساختن جداول تجزیه LALR

- برای مثال، اگر تجزیه‌کننده LR آیتم‌های I_3 و I_6 را برروی پشته قرار دهد، تجزیه‌کننده LALR نیز حالت I_{36} را برروی پشته قرار می‌دهد.
- در حالت کلی هر تجزیه‌کننده LR و LALR معادل آن برای ورودی‌های درست عملیات مشابه انجام می‌دهند.
- وقتی در ورودی خطا وجود داشته باشد، تجزیه‌کننده LALR ممکن است تعداد بیشتری کاهش انجام دهد تا به خطا برسد، اما تجزیه‌کننده LALR هیچ‌گاه عملیات انتقال پس از رسیدن به نقطه خطای تجزیه‌کننده LR انجام نمی‌دهد.

ساختن جداول تجزیه LALR

- برای مثال، برای ورودی $ccd ، تجزیه‌کننده LR حالات 4 3 3 0 را برروی پشته قرار می‌دهد و در حالت ۴ یک خطا تشخیص می‌دهد. تجزیه‌کننده LALR حالات 47 36 36 0 را برروی پشته قرار می‌دهد، اما در حالت ۴۷ با ورودی $ عملیات کاهش $C \rightarrow d$ را انجام می‌دهد و برروی پشته 89 36 36 0 قرار می‌گیرد. سپس یک عملیات کاهش دیگر با استفاده از $C \rightarrow cC$ انجام داده و 89 36 0 برروی پشته قرار می‌گیرد و در نهایت با یک کاهش دیگر 2 0 برروی پشته قرار گرفته می‌شود. در نهایت در حالت ۲ با ورودی $ تجزیه‌کننده خطا صادر می‌کند.

ساختن جداول تجزیه LALR

- الگوریتم سریع‌تری برای ساخت جدول تجزیه LALR وجود دارد که در اینجا به آن نمی‌پردازیم.
- یک زبان برنامه‌نویسی معمول با ۵۰ تا ۱۰۰ ترمینال و حدود ۱۰۰ قانون تولید می‌تواند یک جدول تجزیه با چند صد حالت تولید کند. بسیاری از خانه‌ها در جدول تجزیه تکراری هستند و بنابراین روش‌هایی برای فشرده‌سازی جدول تجزیه وجود دارد.

ساختن جداول تجزیه LALR

- معمولا برای گرامرهای مبهم نمی‌توان تجزیه‌کننده LR تولید کرد، اما برای برخی از گرامرهای مبهم می‌توان جدول تجزیه را به نحوی طراحی کرد که همیشه تصمیم درست در فرایند تجزیه اتخاذ کرد و تنها یک درخت تجزیه تولید کند. در برخی مواقع گرامر مبهم برای توصیف زبان ساده‌تر از معادل غیرمبهم آن است و بنابراین گاه می‌توان در شرایط خاص از گرامرهای مبهم در تجزیه‌کننده‌های LR استفاده کرد.

بازیابی خطا در تجزیه‌کننده LR

- تجزیه‌کننده LR خطاها را با مراجعه به جدول تجزیه شناسایی می‌کند.
- تجزیه‌کننده LR استاندارد به محض وقوع خطا، آن را شناسایی می‌کند، اما تجزیه‌کننده‌های SLR و LALR ممکن است قبل از صدور خطا تعدادی عملیات کاهش انجام دهند.
- در تجزیه‌کننده LR بازیابی خطا با توکن همگام‌کننده [1] به صورت زیر انجام می‌شود.
- پشته بررسی می‌شود تا به حالت s برسیم که با تابع goto به یک متغیر A گذار می‌کند. سپس از تعداد صفر یا بیشتر نمادهای ورودی چشم‌پوشی می‌شود تا اینکه به نماد a برسیم که می‌تواند متغیر A را دنبال کند. سپس حالت Goto(s, A) برروی پشته قرار می‌گیرد و تجزیه ادامه پیدا می‌کند.

[1] panic-mode error recovery

بازیابی خطا در تجزیه‌کننده LR

- ممکن است چند انتخاب برای متغیر A وجود داشته باشد. معمولا متغیری انتخاب می‌شود که نماینده یک قطعه از برنامه باشد برای مثال یک بلوک یا یک عبارت. برای مثال اگر A متغیر stmt باشد آنگاه نماد a می‌تواند نقطه‌ویرگول یا آکولاد بسته باشد که پایان دستور را مشخص می‌کند.

- این روش بازیابی خطا سعی می‌کند عبارتی را که شامل خطای نحوی است حذف کند. تجزیه‌کننده تشخیص می‌دهد که رشته‌ای که از متغیر A به دست می‌آید دارای خطا است. قسمتی از آن رشته پردازش شده است و نتیجه این پردازش تعدادی حالت برروی پشته است. مابقی زیررشته دارای خطا در ورودی است و تجزیه‌کننده سعی می‌کند از قسمتی از ورودی چشم پوشی کند تا به کاراکتری برسد که متغیر A را دنبال می‌کند. با حذف تعدادی حالت از روی پشته و چشم‌پوشی از قسمتی از ورودی و قرار دادن Goto(s, A) برروی پشته، تجزیه‌کننده به احتمال زیاد می‌تواند قسمتی از ورودی که دارای خطاست را پشت سر بگذار و با تجزیه برنامه به صورت عادی ادامه می‌دهد.

بازیابی خطا در تجزیه‌کننده LR

- بازیابی خطا با جایگزینی توکن‌ها [1] بدین صورت پیاده‌سازی می‌شود که هریک از خانه‌های خطا در جدول تجزیه بررسی شده و تشخیص داده می‌شود چه نوع خطاهای رایج برنامه‌نویسی ممکن است در آن مواقع رخ دهد. برای هریک از خطاها در پیام خطای مناسب تهیه می‌شود و حالت‌هایی که باید از پشته حذف شوند و قسمتی از ورودی که باید از آن چشم‌پوشی شود مشخص می‌شوند.
- هریک از خانه‌های خطا در جدول تجزیه با اشاره‌گری به یک تابع مناسب جایگزین می‌شود. این توابع می‌توانند نمادهایی را در ورودی اضافه کنند و یا قسمتی از ورودی را حذف کنند یا تغییر دهند. باید اطمینان حاصل شود که پردازش خطا باعث نمی‌شود تجزیه‌کننده وارد یک حلقه بی‌پایان شود.

[1] phrase-level error recovery