

به نام خدا

# طراحى کامپایلر

آرش شفیعی



## تحليل نحوی

- در این فصل در مورد الگوریتم‌های مختلف تجزیه‌گرامرها 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<sup>۱</sup> یک زبان، نحوه قرارگیری توکن‌ها در جملات زبان را تعیین می‌کند.
- ساختار نحوی یک زبان را می‌توان توسط گرامرهای مستقل از متن توصیف کرد.

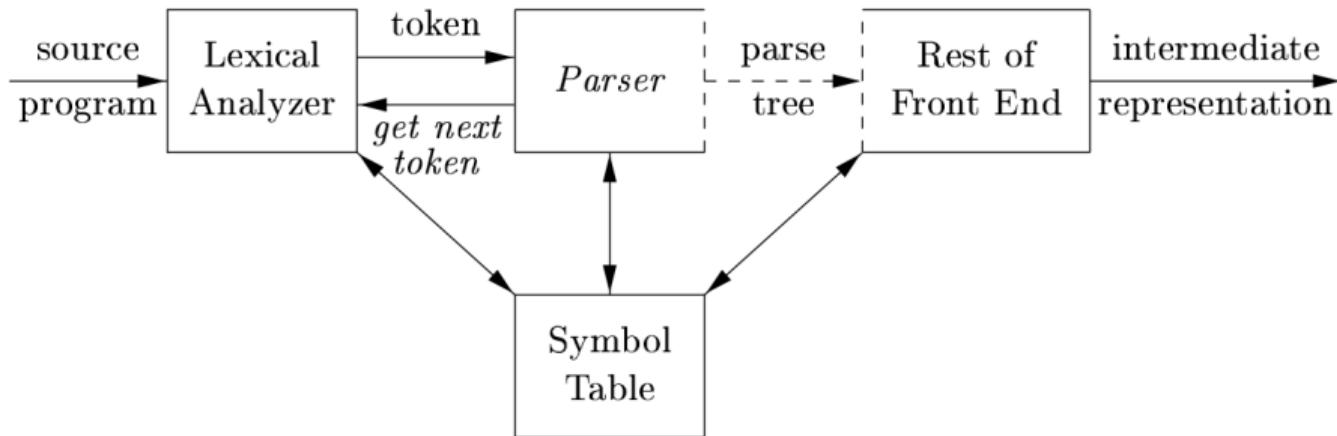
---

<sup>۱</sup> syntax

- 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<sup>۱</sup> یا تجزیه‌کننده<sup>۲</sup> (پارس) توکن‌ها را از تحلیل‌گر لغوی دریافت می‌کند و بررسی می‌کند آیا دنبالهٔ توکن‌های دریافت شده می‌توانند توسط زبان‌گرامر توصیف شده تولید شوند یا خیر.



<sup>۱</sup> syntax analyzer

<sup>۲</sup> parser

# تحلیل نحوی

- یک تجزیه کننده همچنین معمولاً قادر است هر نوع خطای نحوی را گزارش کرده و ادامه رشته ورودی را پس از خطا تجزیه کند.
- 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<sup>۱</sup> ، بالا به پایین<sup>۲</sup> ، و پایین به بالا<sup>۳</sup>.
- الگوریتم‌های تجزیه عمومی مانند الگوریتم سی‌وای‌کا<sup>۴</sup> و الگوریتم ایرلی<sup>۵</sup> 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.

---

<sup>1</sup> universel

<sup>2</sup> top-down

<sup>3</sup> bottom-up

<sup>4</sup> Cocke-Younger-kasami (CYK)

<sup>5</sup> Earley

- 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 توکن‌های 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 همه ساختارهای زبان‌های برنامه‌نویسی موجود کافی هستند.

# 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<sup>1</sup> (*stmt*) و عبارت<sup>2</sup> (*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.

---

<sup>1</sup> statement

<sup>2</sup> 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<sup>۱</sup> 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<sup>۲</sup> 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.

---

<sup>۱</sup> terminal

<sup>۲</sup> nonterminal

# گرامرهای مستقل از متن

۳. یکی از متغیرها به عنوان نماد آغازین<sup>۱</sup> استفاده می‌شود.
۴. قوانین تولید<sup>۲</sup> 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<sup>۳</sup>، یک نماد → که گاهی با =:: نشان داده می‌شود و یک بدنه یا سمت راست<sup>۴</sup> قانون که از صفر یا چند ترمینال و متغیر تشکیل شده است.

---

<sup>۱</sup> start symbol

<sup>۲</sup> production rule

<sup>۳</sup> head or left side

<sup>۴</sup> body or right side

# گرامرهای مستقل از متن

- در فرایند تجزیه 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. مجموعه همه رشته‌هایی که با شروع از نماد آغازین و اعمال قوانین یک گرامر به دست می‌آیند، زبان آن گرامر را تعیین می‌کنند.

## گرامرهای مستقل از متن

- 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.
- با در نظر گرفتن تنها عملگرهای جمع، ضرب، و پرانتز، یک عبارت<sup>۱</sup> به نام E تشکیل شده است از مجموع تعدادی جمله<sup>۲</sup> به نام T که با عملگر + با یکدیگر جمع شده‌اند و هریک از جملات تشکیل شده است از ضرب تعدادی فاکتور (ضریب)<sup>۳</sup> به نام F که با استفاده از عملگر \* در یکدیگر ضرب شده‌اند. هریک از فاکتورها می‌تواند یک شناسه باشد، و یا خود یک عبارت باشد که در بین دو پرانتز قرار گرفته است.

---

<sup>1</sup> expression

<sup>2</sup> term

<sup>3</sup> factor

# گرامرهای مستقل از متن

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

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

# گرامرهاي مستقل از متن

- گرامر زير عبارات رياضي را تجزيه مى کند که شامل عملگرهای + و - و \* و / و ( و ) هستند. کلمه **id** در الواقع نوع توکن شناسه است که در تحليل لغوی استخراج شده است.

<i>expression</i>	$\rightarrow$	<i>expression + term</i>
<i>expression</i>	$\rightarrow$	<i>expression - term</i>
<i>expression</i>	$\rightarrow$	<i>term</i>
<i>term</i>	$\rightarrow$	<i>term * factor</i>
<i>term</i>	$\rightarrow$	<i>term / factor</i>
<i>term</i>	$\rightarrow$	<i>factor</i>
<i>factor</i>	$\rightarrow$	( <i>expression</i> )
<i>factor</i>	$\rightarrow$	<b>id</b>

# گرامرهای مستقل از متن

- در این فصل از علائم و نشانه‌گذاری‌های زیر استفاده می‌کنیم.
- ترمinal‌ها 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 ... و 9 و رشته‌های پرنگ 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  $\alpha$  و  $\beta$  و  $\gamma$  و  $\delta$  نمایش می‌دهیم.
- برای نمایش دنباله‌ای از ترمینال‌ها و متغیرها از حروف یونانی مانند  $\alpha$  و  $\beta$  استفاده می‌کنیم. مثلاً  $\alpha \rightarrow A$  یک قانون گرامر است.
- وقتی یک متغیر چندین بدنی داشته باشد آنها را با علامت خط عمودی از یکدیگر جدا می‌کنیم مثلاً  $A \rightarrow \alpha_1|\alpha_2|\dots|\alpha_k$
- 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.

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

# 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 می‌افتد 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.

- 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 همه 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 خطای بازیابی شده و بررسی برنامه را ادامه دهد تا خطاهای بعدی را تشخیص دهد. همچنین مدیریت خطا نباید سریع 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<sup>۱</sup> : در این روش تجزیه کننده از توکن‌ها یک‌به‌یک چشم پوشی می‌کند تا به یکی از توکن‌های همگام کننده<sup>۲</sup> 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 خطاهای نادیده گرفته شوند اما مزیت آن سادگی پیاده‌سازی آن است.
- بازیابی با جایگزینی توکن‌ها<sup>۳</sup> : با رخداد خطا، تجزیه کننده 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.

---

<sup>1</sup> panic-mode recovery

<sup>2</sup> synchronizing tokens

<sup>3</sup> phrase-level recovery

- قوانین گرامری تشخیص خطا<sup>۱</sup> : با پیش‌بینی کردن خطاهای معمول برنامه‌نویسی می‌توان تعدادی قوانین گرامری به گرامر اضافه کرد که خطاهای را تشخیص می‌دهند.
- تصحیح عمومی و بهینه<sup>۲</sup> : معمولاً انتظار داریم تجزیه‌کننده‌کمترین 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.

<sup>۱</sup> error production rules

<sup>۲</sup> global correction

- به فرایندی که در آن یک رشته توسط قوانین یک گرامر تولید می‌شود، فرایند اشتقاق<sup>۱</sup> گفته می‌شود.
- با شروع از نماد آغازین، در هرگام یکی از متغیرها با بدنی یکی از قوانین متعلق به آن متغیر جایگزین می‌شود. دنباله ترمینال‌ها و متغیرهایی که در هرگام به دست می‌آید را یک صورت جمله‌ای<sup>۲</sup> می‌نامیم. اگر با جایگزین کردن متغیرها در صورت‌های جمله‌ای توسط بدنی 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.

---

<sup>۱</sup> derivation

<sup>۲</sup> sentential form

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

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$$

- فرض کنید می‌خواهیم جمله  $(id)$  – 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  $(id)$  – .

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$$

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

- دنباله‌ای از نمادها به صورت  $\alpha A \beta$  را در نظر بگیرید به طوری که  $\alpha$  و  $\beta$  دنباله‌ای از نمادهای پایانی و غیرپایانی (ترمینال‌ها و متغیرهای) گرامر هستند و  $A$  یک نماد غیرپایانی (متغیر) است.
- فرض کنید  $\gamma \rightarrow A$  یک قانون تولید باشد. آنگاه می‌نویسیم  $\alpha A \beta \Rightarrow \alpha \gamma \beta$ . نماد  $\Rightarrow$  به معنی مشتق کردن در یک گام<sup>۱</sup> است.

---

<sup>1</sup> derives in one step

## اشتقاق

- وقتی دنباله‌ای به صورت  $\alpha_n \dots \alpha_2 \Rightarrow \alpha_1$  داشته باشیم به طوری که  $1 \leq n$ ، می‌گوییم  $\alpha_n$  از  $\alpha_1$  مشتق می‌شود و یا  $\alpha_1$  به دست می‌دهد  $\alpha_n$  و یا  $\alpha_1$  در صفر یا چندگام مشتق می‌کند  $\alpha_n$ . به عبارت دیگر:

۱. به ازای هر صورت جمله‌ای  $\alpha$  داریم  $\alpha \xrightarrow{*} \alpha$  یعنی هر صورت جمله‌ای می‌تواند خود را در صفر یا چندگام<sup>۱</sup> مشتق کند.

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

- همچنین گاهی می‌نویسیم  $\xrightarrow{+}$  به معنی مشتق کردن در یک یا چندگام.

- اگر  $\alpha \xrightarrow{*} S$  جایی که  $S$  نماد آغازین گرامر  $G$  است، می‌گوییم  $\alpha$  یک صورت جمله‌ای<sup>۲</sup> از گرامر  $G$  است.

- یک صورت جمله‌ای شامل متغیرها و ترمینال‌هاست. یک جمله<sup>۳</sup> از یک گرامر یک صورت جمله‌ای است که در آن هیچ متغیری نباشد.

---

<sup>1</sup> derives in zero or more steps

<sup>2</sup> sentential form

<sup>3</sup> sentence

- زبان تولید شده توسط یک گرامر مجموعه‌ای است از همه 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  $w \xrightarrow{*} S$ .
- 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.

- گرامر زیر را در نظر بگیرید.

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$$

- جمله از این گرامر است زیرا فرایند اشتقاق زیر برای آن وجود دارد :

$$E \Rightarrow -E \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$$

- می‌نویسیم  $E \xrightarrow{*} -(id + id)$  و می‌خوانیم جمله 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.

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

۱. در اشتقاق چپ<sup>۱</sup> ، متغیری که در صورت جمله‌ای در سمت چپ بقیه متغیرها قرار دارد و به عبارت دیگر چپ‌ترین<sup>۲</sup> است، انتخاب می‌شود. اگر  $\alpha \Rightarrow \beta$  گامی باشد که در آن چپ‌ترین متغیر  $\alpha$  انتخاب شود، می‌نویسیم  $\alpha \xrightarrow{lm} \beta$ .
۲. در اشتقاق راست<sup>۳</sup> ، متغیری که راست‌ترین<sup>۴</sup> است انتخاب می‌شود و می‌نویسیم  $\alpha \xrightarrow{rm} \beta$ .

<sup>۱</sup> leftmost derivation

<sup>۲</sup> leftmost

<sup>۳</sup> rightmost derivation

<sup>۴</sup> rightmost

- برای مثال :

$$E \xrightarrow{rm} -E \xrightarrow{rm} -(E) \xrightarrow{rm} -(E+E) \xrightarrow{rm} -(E+id) \xrightarrow{rm} -(id+id)$$

- به طور خلاصه می‌گوییم  $wA\gamma \xrightarrow{l_m} w\delta\gamma$  جایی که  $w$  فقط از ترمینال‌ها تشکیل شده و  $\delta \rightarrow A$  یک قانون تولید است و  $\gamma$  رشته‌ای است تشکیل شده از متغیرها و ترمینال‌ها.

- اگر  $S \xrightarrow{l_m} \alpha$  آنگاه می‌گوییم  $\alpha$  یک صورت جمله‌ای چپ<sup>۱</sup> از گرامر است.

---

<sup>1</sup> left sentential form

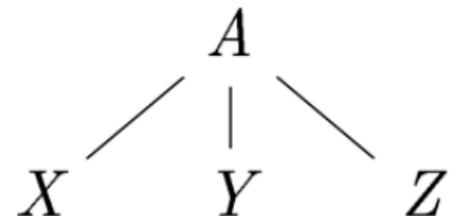
- درخت تجزیه<sup>۱</sup> یک نمایش گرافیکی از فرایند تجزیه است که در آن ترتیب جایگزینی متغیرها نشان داده نمی‌شود.
- هر رأس میانی در درخت تجزیه، اعمال یک قانون در فرایند اشتاقاً را نشان می‌دهد. اگر یک رأس با بروچسب  $A$  در درخت تجزیه داشته باشیم، فرزندان آن از سمت چپ به راست به ترتیب ترمینال‌ها و متغیرهایی هستند که در بدنه یکی از قوانین متعلق به  $A$  قرار دارند.

---

<sup>۱</sup> parse tree

- یک درخت تجزیه نشان می‌دهد چگونه با شروع از نماد آغازین یک گرامر می‌توان یک رشته از یک زبان را مشتق کرد.
- توجه کنید در نظریه زبان‌ها یک زبان مجموعه‌ای از رشته‌های است و یک گرامر برای یک زبان مشخص می‌کند آیا یک رشته متعلق به یک زبان است یا خیر. در علم کامپایلر یک زبان برنامه‌نویسی مجموعه‌ای است از همه برنامه‌هایی که توسط گرامر آن زبان مشتق شوند. بنابراین رشته به دست آمده با شروع از متغیر آغازین یک گرامر، درواقع یک برنامه در آن زبان است.

- اگر متغیر A قانون  $XYZ \rightarrow A$  را در یک گرامر داشته باشد، آنگاه درخت تجزیه آن می‌تواند شامل یک رأس A باشد که فرزندان آن به ترتیب از سمت چپ X و Y و Z هستند.



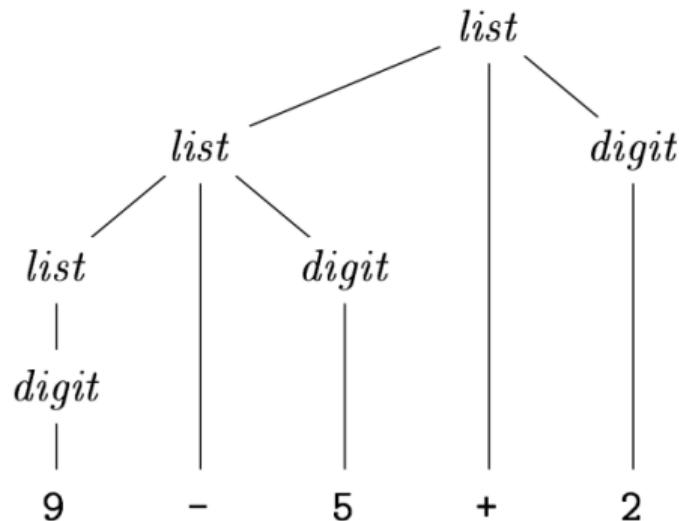
- یک درخت تجزیه<sup>۱</sup> برای یک گرامر مستقل از متن درختی است که ویژگی‌های زیر را دارد.
- ۱ ریشه آن با متغیر آغازین برچسب زده است.
- ۲ هر یک از برگ‌های آن با یک ترمینال یا رشته تهی برچسب زده شده‌اند.
- ۳ هر یک از رئوس میانی آن با یک متغیر برچسب زده است.
- ۴ اگر A یک متغیر باشد و  $X_1$  و  $X_2$  و ... و  $X_n$  به ترتیب از سمت چپ فرزندان آن باشند، آنگاه باید قانون تولید  $X_n \dots X_1 \rightarrow A$  وجود داشته باشد. در اینجا هر یک از  $X_1$  تا  $X_n$  ها می‌توانند یک ترمینال یا یک متغیر باشند. اگر قانون  $\epsilon \rightarrow A$  وجود داشته باشد، آنگاه A تنها یک فرزند دارد که با  $\epsilon$  برچسب زده است.

---

<sup>۱</sup> parse tree

## درخت تجزیه

- برگ‌های درخت تجزیه به ترتیب از چپ به راست رشته‌ای را تشکیل می‌دهند که از گرامر مشتق می‌شود (تولید می‌شود یا به دست می‌آید).



- در شکل بالا رشته  $2 + 5 - 9$  از درخت تجزیه مشتق می‌شود.

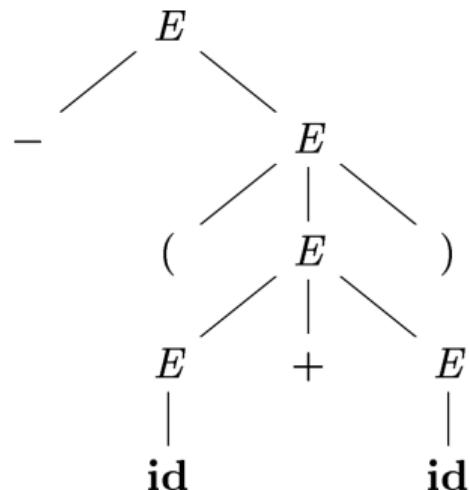
- یک زبان مجموعه‌ای است از همه رشته‌هایی که هرگدام توسط یک درخت تجزیه مشتق می‌شوند. تجزیه فرایندی برای پیدا کردن یک درخت تجزیه توسط یک گرامر برای یک رشته است.

## درخت تجزیه

- گرامر زیر را در نظر بگیرید.

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

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



- 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<sup>۱</sup> 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<sup>۲</sup> نیز گفته می‌شود. به درخت تجزیه‌ای که محصول آن یک جمله باشد، درخت تجزیه کامل<sup>۳</sup> گفته می‌شود.

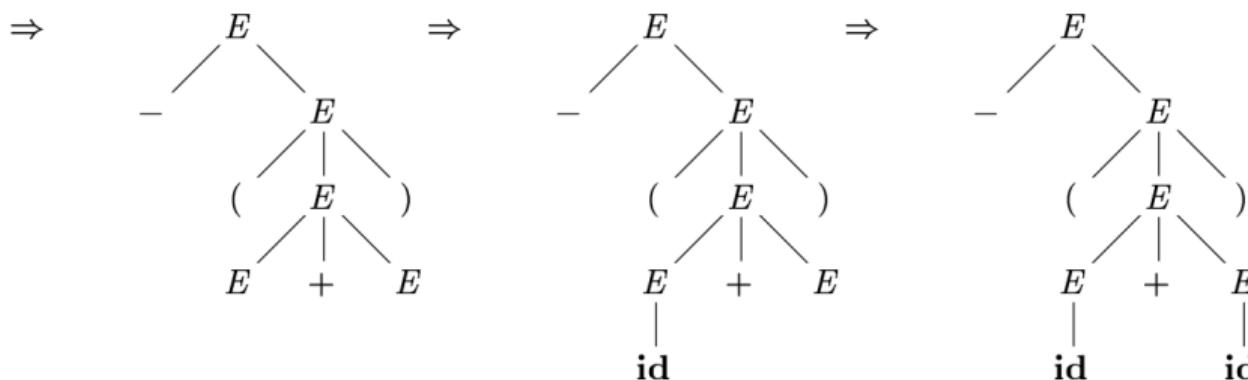
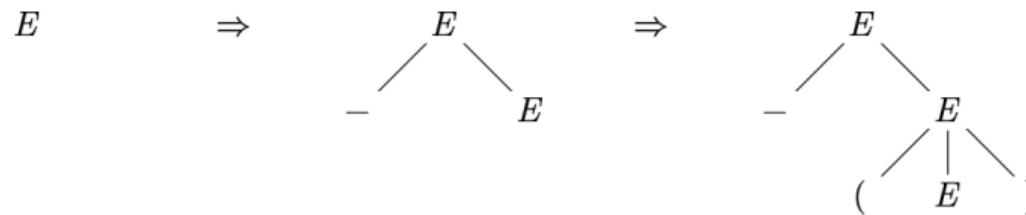
---

<sup>۱</sup> yield

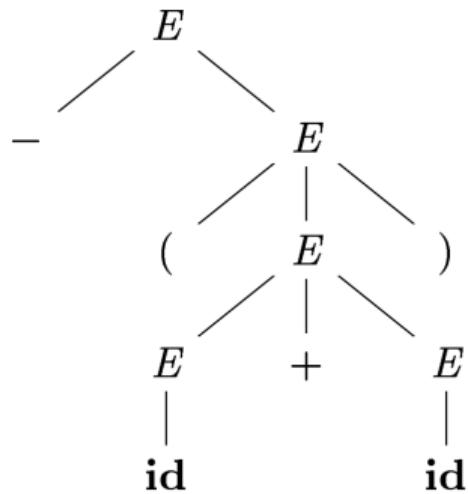
<sup>۲</sup> partial parse tree

<sup>۳</sup> complete parse tree

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



- 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 تجزیه زیر می‌تواند توسط یک اشتقاق چپ یا یک اشتقاق راست تولید شده باشد.



- گرامری که بیش از یک درخت تجزیه برای یک جمله تولید کند، مبهم<sup>۱</sup> نامیده می‌شود.
- به عبارت دیگر یک گرامر مبهم برای تولید یک رشته بیش از یک فرایند اشتقاء چپ (یا بیش از یک فرایند اشتقاء راست) دارد.

---

<sup>۱</sup> ambiguous

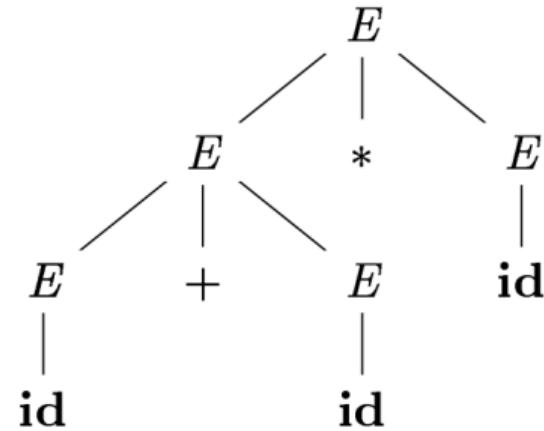
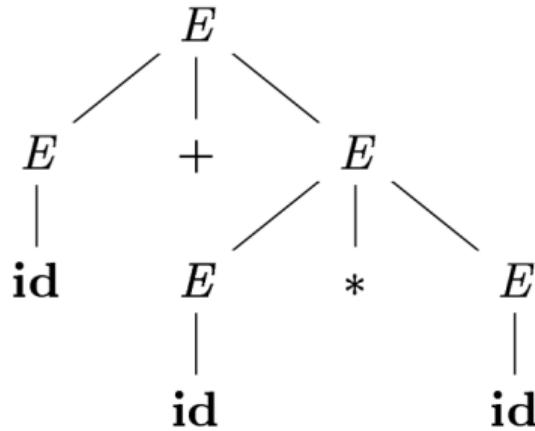
گرامر زیر را در نظر بگیرید.

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

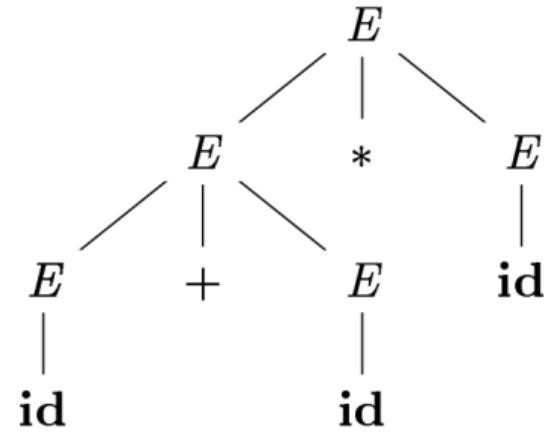
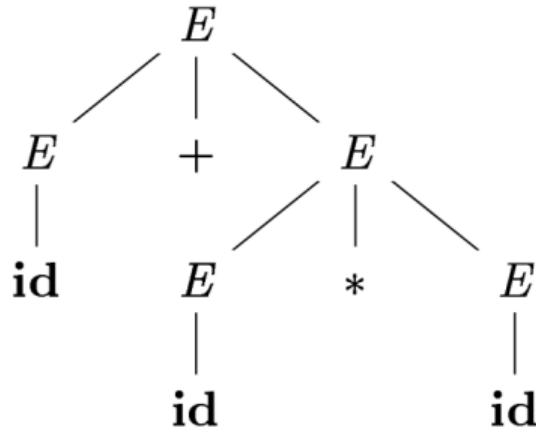
برای به دست آوردن رشته  $\text{id} + \text{id} * \text{id}$  توسط این گرامر دو فرایند اشتقاق چپ وجود دارد.

$$\begin{array}{ll} E \Rightarrow E + E & E \Rightarrow E * E \\ \Rightarrow \text{id} + E & \Rightarrow E + E * E \\ \Rightarrow \text{id} + E * E & \Rightarrow \text{id} + E * E \\ \Rightarrow \text{id} + \text{id} * E & \Rightarrow \text{id} + \text{id} * E \\ \Rightarrow \text{id} + \text{id} * \text{id} & \Rightarrow \text{id} + \text{id} * \text{id} \end{array}$$

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



- 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  $(b * c) + a$  و درخت سمت راست معادل  $c * (a + b)$  خواهد بود.



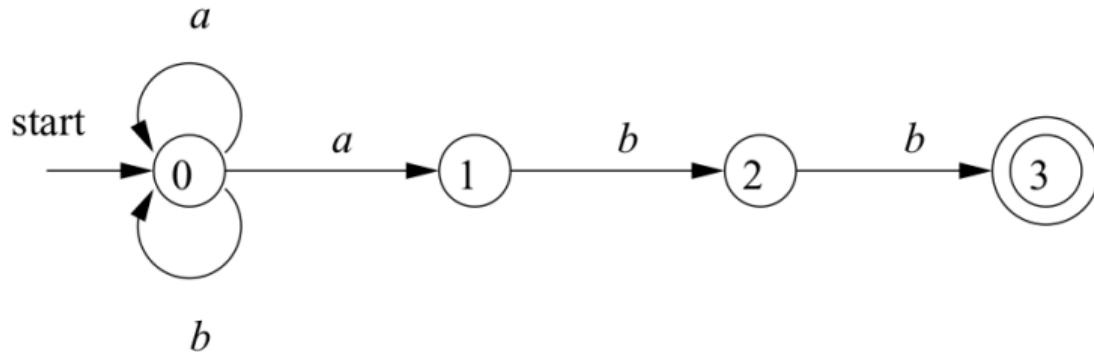
## گرامرهای منظم

- 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  $a|b)^*abb(a|b)^*$  را می‌توان توسط گرامر منظم زیر توصیف کرد.

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

# 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  $n$  با ورودی  $a$  به حالت  $j$  می‌رود آنگاه قانون  $j \rightarrow aA_i \rightarrow aA_j$  را به گرامر اضافه می‌کنیم. اگر حالت  $n$  با ورودی  $\epsilon$  به حالت  $j$  می‌رود آنگاه قانون  $j \rightarrow A_i \rightarrow A_j$  را به گرامر اضافه می‌کنیم.
۳. اگر حالت  $n$  یک حالت نهایی است آنگاه قانون  $\epsilon \rightarrow A_i$  را اضافه می‌کنیم.
۴. اگر حالت  $n$  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  $\{a^n b^n | n \geq 1\} = L$  زبانی است که نمی‌توان برای توصیف آن از یک ماشین متناهی استفاده کرد. توسط لم تزریق اثبات می‌شود که این زبان متعلق به دستهٔ 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.

## گرامرهاي 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 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 نسبتاً ساده‌اند و با عبارت‌های منظم ساده تولید می‌شوند و نیازی به افزودن گرامرهاي پيچيده برای آنها وجود ندارد. به علاوه روشی وجود دارد که تحلیل‌گر لغوی مستقیماً از عبارت منظم تولید می‌شود. به این دلایل تحلیل‌گر لغوی از تحلیل‌گر نحوی جدا می‌شود.

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

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

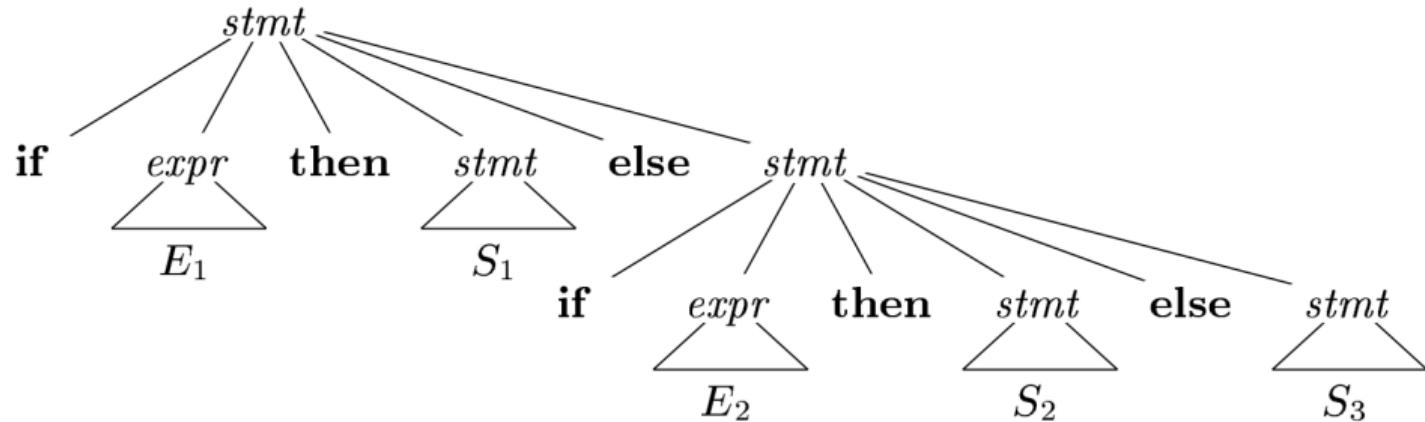
- در اینجا **other** به معنی هر دستور دیگری به غیر از دستورات شرطی if-else است.

# رفع ابهام

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

**if**  $E_1$  **then**  $S_1$  **else** **if**  $E_2$  **then**  $S_2$  **else**  $S_3$

- برای این جمله درخت تجزیه زیر وجود دارد.



## رفع ابهام

- حال جمله زیر را در نظر بگیرید.

**if**  $E_1$  **then** **if**  $E_2$  **then**  $S_1$  **else**  $S_2$

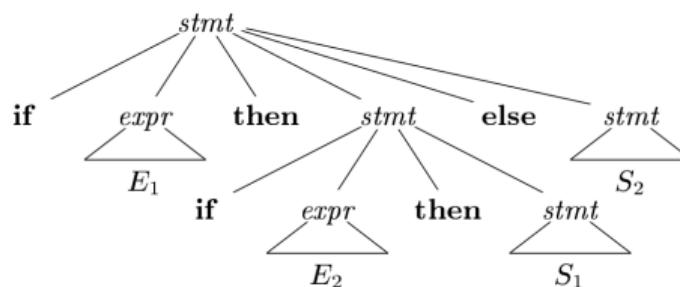
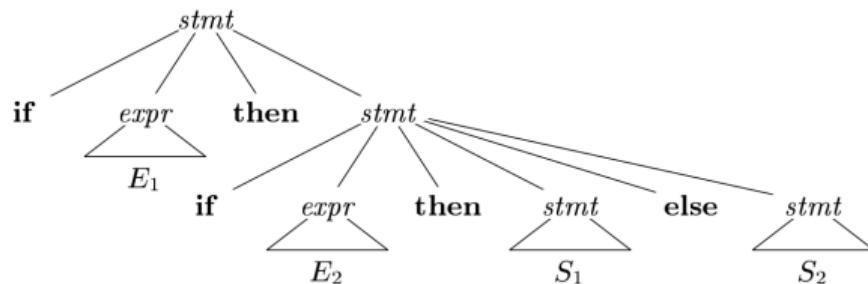
- درخت تجزیه‌ای برای این جمله رسم کنید.

# رفع ابهام

- حال جمله زیر را در نظر بگیرید.

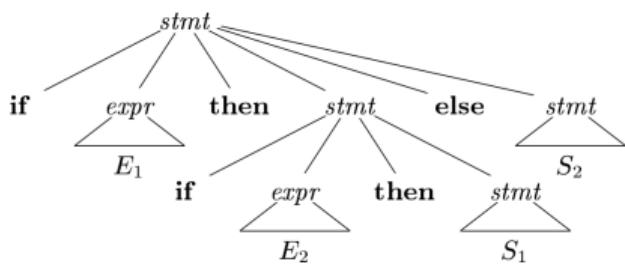
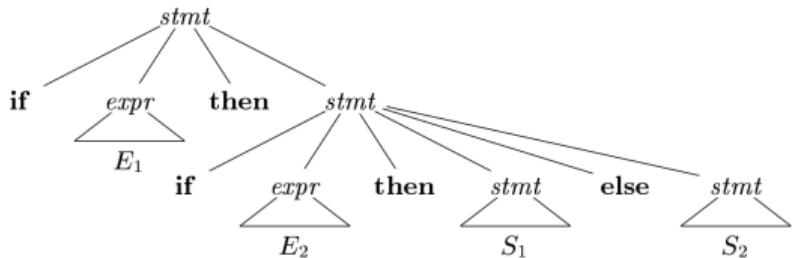
**if**  $E_1$  **then** if  $E_2$  **then**  $S_1$  **else**  $S_2$

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



# رفع ابهام

- در همه 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.

- 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.

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

```

stmt   → matched_stmt
      | open_stmt
matched_stmt → if expr then matched_stmt else matched_stmt
      | other
open_stmt   → if expr then stmt
      | if expr then matched_stmt else open_stmt
  
```

## وابستگی عملگرها

- به طور قراردادی  $2 + 5 + 9$  معادل است با  $2 + (5 + 9)$  و عبارت  $2 - 5 - 9$  معادل است با  $(2 - 5) - 9$

- وقتی عملوند 5 دو عملگر + در سمت چپ و راست خود دارد، به طور قراردادی ابتدا عملگر سمت چپ را اعمال می‌کنیم. می‌گوییم عملگر + وابسته چپ<sup>1</sup> است، زیرا عملوند 5 با دو عملگر + در سمت چپ و راست متعلق به عملگر سمت چپ است. در بیشتر زبان‌ها برنامه‌نویسی همه عملگرهای حسابی جمع و تفریق و ضرب و تقسیم وابسته چپ هستند.

---

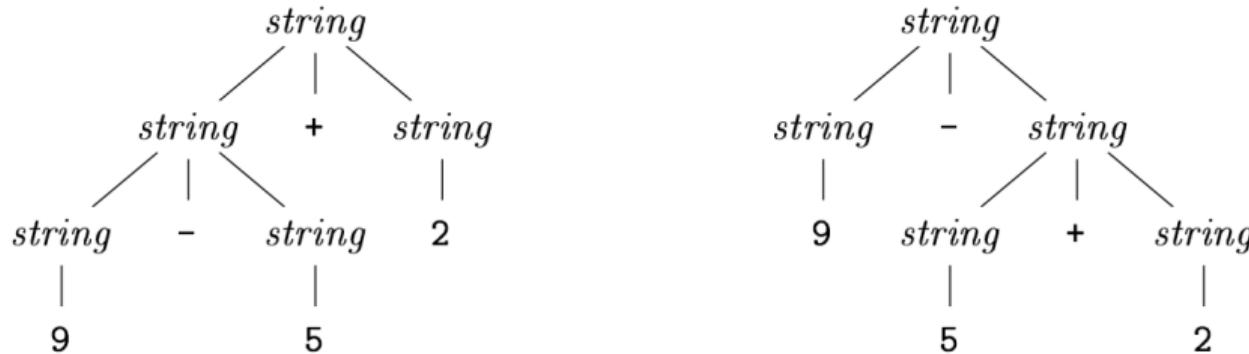
<sup>1</sup> left-associative

- فرض کنید گرامری به صورت زیر داریم :

string → string + string | string - string | 0|1|2|3|4|5|6|7|8|9

# وابستگی عملگرها

- حال عبارت  $2 + 5 - 9$  را در نظر بگیرید. برای این عبارت دو درخت تجزیه به صورت زیر وجود دارد.



- درخت سمت چپ در واقع  $2 + (5 - 9)$  محاسبه می‌کند در حالی که درخت سمت راست  $(2 + 5) - 9$  را محاسبه می‌کند. دو درخت تجزیه دو معنای متفاوت دارند.

- درخت تجزیه سمت راست مقدار 2 را برای این عبارت محاسبه می‌کند درحالی که مقدار عبارت برابر است با . 6

# وابستگی عملگرها

- گرامر قبلی را می‌توانیم به صورت زیر رفع ابهام کنیم.

- گرامر زیر ساختار نحوی عبارات ریاضی که از عملگر جمع و تفریق (با وابستگی چپ) تشکیل شده‌اند را توصیف می‌کند.

---

list  $\rightarrow$  list + digit

list  $\rightarrow$  list - digit

list  $\rightarrow$  digit

digit  $\rightarrow$  ۰|۱|۲|۳|۴|۵|۶|۷|۸|۹

---

# وابستگی عملگرها

- برخی از عملگرها مانند عملگر توان وابسته راست هستند. برای مثال  $4^2 \cdot 3^4$  معادل است با  $(4^3)^2$ .
- به عنوان یک مثال دیگر، عملگر انتساب = در زبان سی وابسته راست است، یعنی  $a = b = c$  معادل است با  $.a = (b = c)$ .
- رشته‌هایی که از عملگر وابسته راست انتساب تشکیل شده‌اند با استفاده از گرامر زیر به دست می‌آید.

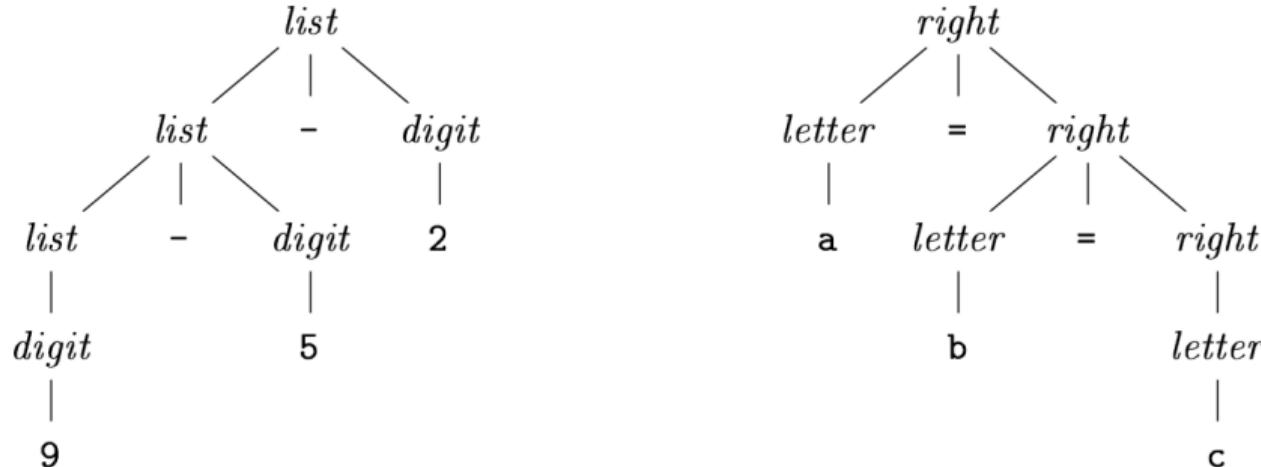
---

right → letter = right | letter  
letter → a | b | ... | z

---

# وابستگی عملگرها

- در شکل زیر درخت تجزیه برای عبارت  $2 - 5 - 9$  با عملگر وابسته چپ تفريق و درخت تجزیه برای عبارت  $c = b = a$  با عملگر وابسته راست انتساب نشان داده شده‌اند.



- عبارت  $2 * 5 + 9$  را در نظر بگیرید. این عبارت را می‌توانیم به دو صورت  $2 * (5 + 9)$  و  $(2 * 5) + 9$  تفسیر کنیم. قوانین وابستگی بر روی عملگرهای همنوع (عملگرهایی که تقدم یکسانی دارند) اعمال می‌شوند، اما عملگر  $*$  و  $+$  دو عملگر متفاوت هستند.
- به طور قراردادی عملگر  $*$  اولویت یا تقدم بالاتری<sup>۱</sup> نسبت به عملگر  $+$  دارد. بنابراین عبارت  $2 * 5 + 9$  به صورت  $(2 * 5) + 9$  تفسیر می‌شود.

<sup>1</sup> higher precedence

- یک عبارت محاسباتی را می‌توانیم بر اساس یک جدول تقدم و وابستگی محاسبه کنیم. در جدول تقدم و وابستگی تعیین شده است که + و - تقدم یکسانی دارند و وابستگی از چپ دارند. عملگرهای \* و / تقدم یکسانی دارند و تقدم آنها از + و - بیشتر است. همچنین وابستگی آنها از چپ است.

\* / : left-associative

+ - : left-associative

- برای حفظ تقدم عملگرها از یک متغیر اضافی در گرامر استفاده می‌کنیم.
- توجه کنید که عملگرهایی که تقدم بالاتری دارند در درخت تجزیه در سطوح پایین‌تر قرار می‌گیرند. بنابراین گرامر باید عملگرهای با تقدم پایین را زودتر تجربه کند.
- برای تجزیه جمع و تفریق از قوانین زیر استفاده می‌کنیم.

---

$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$

---

- از آنجایی که جمع و تفریق وابستهٔ چپ هستند عملگرهای جمع سمت راست عبارت باید زودتر تجزیه شوند. قانون  $\text{expr} \rightarrow \text{expr} + \text{term}$  عملگرهای جمع در سمت راست عبارت را زودتر تجزیه می‌کند و در نتیجه وابستگی چپ ایجاد می‌کند. به طور کلی هنگامی که متغیر سمت چپ یک قانون در سمت چپ بدنه قانون قرار بگیرد وابستگی چپ ایجاد می‌شود.

- پس از تجزیه جمع و تفریق، عملگرهای ضرب و تقسیم را به صورت زیر تجزیه می‌کنیم.

---

$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{term} / \text{factor} \mid \text{factor}$

---

- پرانتر در بالاترین اولویت قرار دارد و بنابراین باید در آخرین مرحله تجزیه شود، بنابراین قانون زیر را برای تجزیه پرانترها اضافه می‌کنیم.

---

$\text{factor} \rightarrow \text{digit} \mid (\text{expr})$

---

- برای افزودن عملگرهای دیگر به این گرامر می‌توانیم یک متغیر به ازای یک دسته از عملگرها با اولویت بالاتر یا پایین‌تر بیافزاییم.

- پس به طور خلاصه برای یک عبارت محاسباتی گرامر زیر را خواهیم داشت.

---

```
expr → expr + term | expr - term | term  
term → term * factor | term / factor | factor  
factor → digit | (expr)  
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

---

- عبارت  $6 + 3 * 2 / (3 + 7) - (4 * 2 - 1)$  را تجزیه کنید.

یک زیر مجموعه از دستورات زبان جاوا را می توانیم توسط گرامر زیر تجزیه کنیم.

```
stmt → id = expr ;  
| if (expr) stmt  
| if (expr) stmt else stmt  
| while (expr) stmt  
| do stmt while (expr);  
| {stmts}  
  
stmts → stmts stmt | ε
```

## تجزیه بالا به پایین

- قبل از بررسی مفصل تجزیه‌کننده بالا به پایین، برای یک گرامر ساده که زیر مجموعه‌ای از گرامر زبان جawa و سی است، یک تجزیه‌کننده بالا به پایین<sup>۱</sup> می‌سازیم و سپس در مورد روند کلی ساختن تجزیه‌کننده بالا به پایین صحبت می‌کنیم.
- گرامر زیر را در نظر بگیرید.

```
stmt   →  expr ;
        |  if ( expr ) stmt
        |  for ( optexpr ; optexpr ; optexpr ) stmt
        |  other
```

```
optexpr →  ε
        |  expr
```

- 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.

---

<sup>1</sup> top-down parser

## تجزیه بالا به پایین

- تجزیه کننده بالا به پایین یک درخت تجزیه با یک ریشه می‌سازد به طوری که برچسب ریشه درخت متغیر آغازین گرامر (*stmt*) است.
- تجزیه کننده 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) بدنه قانون انتخاب شده قرار می‌دهد.
  ۲. رأس بعدی در درخت تجزیه که با یک متغیر برچسب زده است و چپ‌ترین متغیر در بین همه برگ‌های انتخاب می‌کند و آن برگ را با توجه به رشته ورودی گسترش می‌دهد.
  - در هرگام از فرایند تجزیه، تجزیه کننده با توجه به توکن بعدی<sup>۱</sup> در رشته ورودی تصمیم می‌گیرد چه قانونی را انتخاب کند.

---

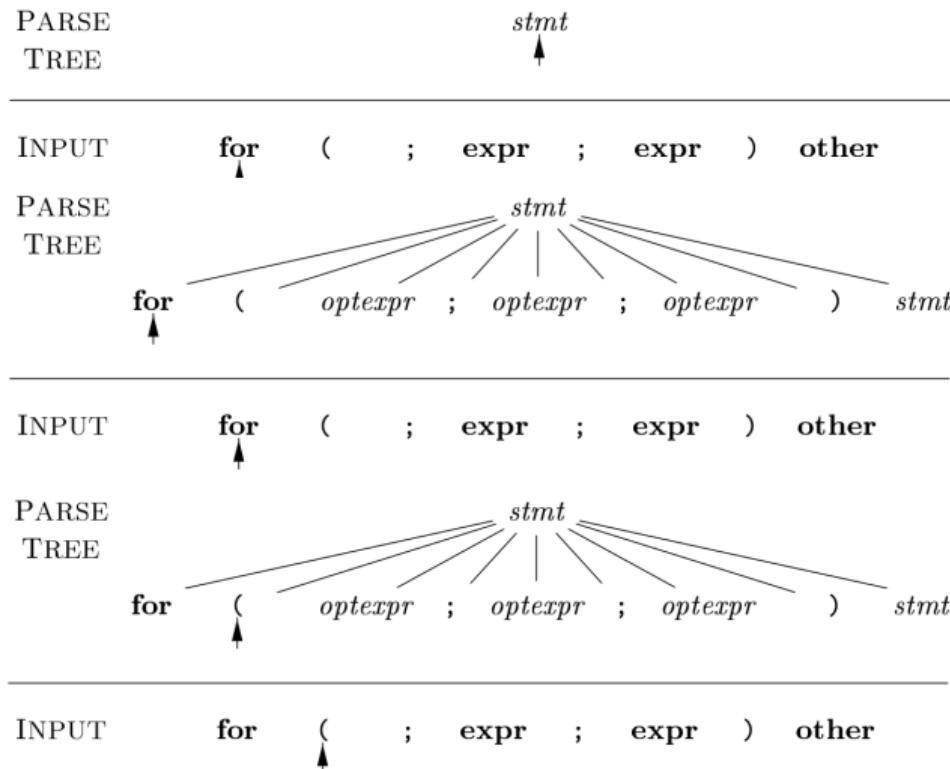
<sup>1</sup> lookahead

## تجزیه بالا به پایین

- در تجزیه رشته `for ( ;expr;expr) other` اولین توکن، واژه `for` است. بنابراین ریشه درخت تجزیه که با `stmt` برچسب زده شده است با قانونی از متغیر `stmt` گسترش می‌یابد که بدنه آن با واژه `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<sup>۱</sup> قانون بعدی انتخاب شود.
- معمولاً چنین عقبگردهایی پرهزینه است و به دنبال روش‌های تجزیه‌ای هستیم که از چنین عقبگردهایی جلوگیری کنند.

---

<sup>1</sup> backtrack

## تجزیه پیش‌بینی کننده

- تجزیه کاهشی بازگشتی<sup>۱</sup> روشی بالا به پایین برای تحلیل نحوی است که در آن مجموعه‌ای از توابع بازگشتی برای پردازش رشته ورودی استفاده می‌شوند. در این تجزیه کننده، به ازای هریک از متغیرهای گرامر یک تابع در نظر گرفته می‌شود.
- یکی از انواع ساده تجزیه کاهشی بازگشتی، تجزیه پیش‌بینی کننده<sup>۲</sup> است. در تجزیه پیش‌بینی کننده از پسگرد جلوگیری می‌شود.

---

<sup>۱</sup> recursive-descent parsing

<sup>۲</sup> predictive parsing

## تجزیه پیش‌بینی کننده

- یک تجزیه کنندهٔ پیش‌بینی 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<sup>۱</sup> یک ترمینال در گرامر و یک کلمه از رشته ورودی، تجزیه کننده صرفاً 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.

---

<sup>۱</sup> match

## تجزیه پیش‌بینی کننده

- در تجزیه کننده پیش‌بینی کننده‌ای که طراحی کردیم، در پیاده‌سازی تابع  $\text{optexpr}()$  در صورتی که تطبیق رخ ندهد خطایی صادر نکردیم. با این کار در واقع قانون  $\epsilon \rightarrow \text{optexpr}$  را پیاده‌سازی کردیم.
- در حالت کلی اگر قوانینی به صورت  $\epsilon | X_1 | X_2 | \dots | A$  داشته باشیم، در پیاده‌سازی تابع  $A()$  هیچ خطایی صادر نمی‌کنیم، زیرا ممکن است رشته ورودی، در بدنه هیچ یک از قوانین  $A$  منطبق نشود که در اینصورت قانون تهی اعمال می‌شود.

## تجزیه پیش‌بینی کننده

- تجزیه کننده پیش‌بینی کننده برای یک گرامر LL می‌تواند مورد استفاده قرار بگیرد.
- برای تعریف گرامر LL، تابع  $\text{First}(\alpha)$  را به صورت زیر تعریف می‌کنیم، جایی که  $\alpha$  دنباله‌ای از ترمینال‌ها و متغیرهاست.
  - اگر اولین کلمه در دنباله  $\alpha$  ترمینال  $t$  باشد، آنگاه  $\text{First}(\alpha) = \{t\}$
  - اگر اولین کلمه در دنباله  $\alpha$  متغیر  $A$  باشد و داشته باشیم  $A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$  آنگاه  $\text{First}(\alpha) = \text{First}(\beta_1) \cup \text{First}(\beta_2) \cup \dots \cup \text{First}(\beta_n)$

## تجزیه پیش‌بینی کننده

- حال فرض کنید در گرامر  $G$  داشته باشیم  $\alpha \rightarrow A \rightarrow \beta$ . یکی از شروط لازم برای اینکه گرامر  $G$  LL باشد، این است که به ازای هر دو بدنی قانون  $\alpha$  و  $\beta$  متعلق به متغیر  $A$  داشته باشیم  $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$ . در مورد شروط دیگر بعدها صحبت خواهیم کرد.
- از تجزیه کننده پیش‌بینی کننده تنها زمانی می‌توان استفاده کرد که یک گرامر LL باشد. در اینصورت در زمان خطی یک رشته را می‌توان تجزیه کرد.

## تجزیه پیش‌بینی کننده

- برای پیاده‌سازی یک تجزیه کننده پیش‌بینی کننده برای یک گرامر  $LL$  ، 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  $X_n \rightarrow X_1 | X_2 | \dots | X_i \rightarrow A$  است، قانون  $X_i \rightarrow A$  را انتخاب می‌کند، اگر توکن بعدی در رشته ورودی در مجموعه  $First(X_i)$  باشد.
- با فرض برواینکه  $X_i$  دنباله‌ای از ترمینال‌های  $t$  و متغیرهای  $X$  است، به ازای هر ترمینال  $t$ ، توکن بعدی در رشته ورودی باید برابر با ترمینال  $t$  باشد و به ازای هر متغیر  $X$ ، تابع  $(X)$  فراخوانی می‌شود.
- اگر رشته ورودی بدون خطا پایان رسد، رشته متعلق به زبان آن گرامر است.

## تجزیه پیش‌بینی کننده

- ممکن است یک تجزیه کننده کاہشی بازگشتی<sup>۱</sup> در یک حلقه بی‌پایان بیافتد.
- فرض کنید یک قانون بازگشتی چپ<sup>۲</sup> به صورت زیر داشته باشیم :

---

$\text{expr} \rightarrow \text{expr} + \text{term}$

---

<sup>۱</sup> recursive-descent parser

<sup>۲</sup> left-recursive rule

## تجزیه پیش‌بینی کننده

- در این قانون، متغیر سمت چپ قانون برابر با نماد سمت چپ در بدنه قانون است.
- حال در فرایند تجزیه اگر تابع `()` فراخوانی شود، این تابع نیز مجدداً تابع `()` 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 بازگشتهای چپ<sup>۱</sup> است اگر به ازای متغیر  $A$  و صورت جمله‌ای دلخواه  $\alpha$  فرایند اشتقاء  $A \stackrel{+}{\Rightarrow} A\alpha$  وجود داشته باشد.
- تجزیه کننده‌های بالا به پایین نمی‌توانند گرامرهایی که دارای بازگشت چپ هستند را تجزیه کنند، بنابراین بازگشت چپ<sup>۲</sup> باید در گرامر حذف شود.

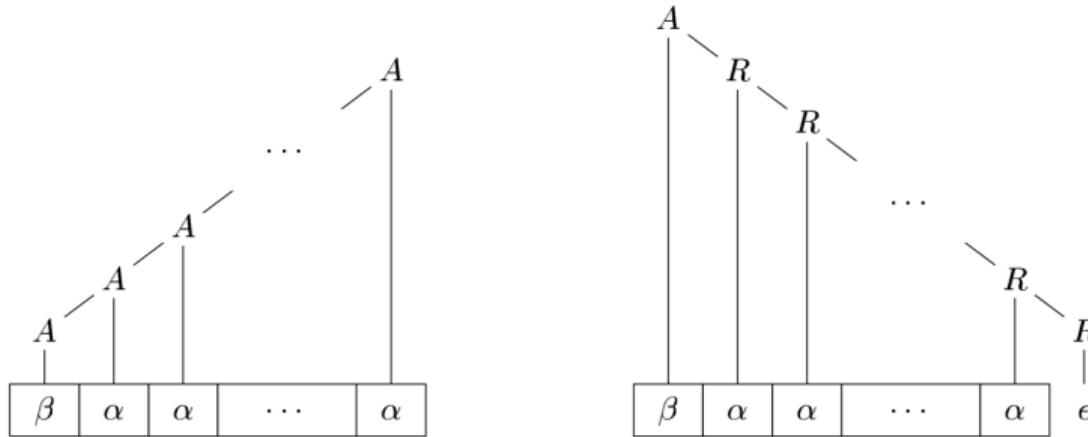
---

<sup>۱</sup> left recursive

<sup>۲</sup> left recursion

## حذف بازگشتی چپ

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



گرامر جایگزین یک گرامر بازگشتی راست<sup>1</sup> است زیرا قانون  $R \rightarrow \alpha R$  را در سمت راست بدنهٔ قانون دارد. قوانین بازگشتی راست در تجزیهٔ کننده‌های بالا به پایین مشکلی به وجود نمی‌آورند.

<sup>1</sup> right recursive

## حذف بازگشت چپ

- در حالت کلی، بازگشت چپ بلافاصله<sup>۱</sup> می‌تواند توسط روش زیر حذف شود.
- ابتدا قوانین را به صورت زیر مرتب می‌کنیم:

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

به طوری که  $\beta_i$  ها با  $A$  آغاز نمی‌شوند.

- سپس قوانین تولید متغیر  $A$  را با قوانین زیر جایگزین می‌کنیم:

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

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

---

<sup>1</sup> immediate left recursion

## حذف بازگشت چپ

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

$$\begin{aligned}S &\rightarrow Aa \mid b \\A &\rightarrow Ac \mid Sd \mid \epsilon\end{aligned}$$

- 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  $A \xrightarrow{+} A$  اتفاق نمی‌افتد و همچنین در آنها قانون تولید تهی یعنی  $\epsilon \rightarrow A$  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  $A_1, A_2, \dots, A_n$ .
- 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  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ , where  
                     $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  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  $A_1 \rightarrow A_1\alpha$  و یا  $A_1 \rightarrow A_m\alpha$  به طوری که  $m > 1$ . در حالت اول در خط ۶ الگوریتم بازگشت چپ بลาواسطه حذف می‌شود. در حالت دوم قانون به همان صورت باقی خواهد ماند.
- به ازای  $i = 2$  سه حالت وجود دارد. یا  $A_2 \rightarrow A_2\alpha$  و یا  $A_2 \rightarrow A_1\alpha$  یا  $A_2 \rightarrow A_m\alpha$  به ازای  $m > 2$ . در حالت اول بازگشت چپ بลาواسطه حذف می‌شود. در حالت دوم قانون به صورت  $A_2 \rightarrow A_p\alpha$  به ازای  $p > 1$  بازنویسی می‌شود و در صورتی که  $p = 2$  باشد، بازگشت چپ بลาواسطه حذف می‌شود. در حالت سوم قانون به همان صورت باقی می‌ماند. پس در پایان در هر صورت خواهیم داشت  $A_2 \rightarrow A_m\alpha$  به ازای  $m > 2$ .
- در حالت کلی پس از اتمام تکرار  $i$  ام در الگوریتم، به ازای همه قوانین  $A_i \rightarrow A_m\alpha$  خواهیم داشت  $A_i \rightarrow A_m\alpha$  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  $\epsilon \rightarrow A$  را حذف می‌کنیم. اگر قانون  $\epsilon \rightarrow A$  وجود داشته باشد، قانون  $S \rightarrow AS\alpha$  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  $A \xrightarrow{+} A$  داشته باشیم که قوانین یکه به صورت  $A \rightarrow B$  وجود داشته باشند.
- بنابراین همه قوانین تولید یکه به صورت  $B \rightarrow A$  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  $i > j$ .
  - در بازنویسی قانون  $A_i \rightarrow A_j$  خواهیم داشت  $A_j \rightarrow A_i$ .
- 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  $R| \epsilon \rightarrow R$  که همچنان بازگشتی چپ است.

## حذف بازگشت چپ

- می خواهیم بازگشت چپ را در گرامر زیر حذف کنیم. قانون تولید تهی در اینجا مشکلی در اجرای الگوریتم ایجاد نمی کند، اما در حالت کلی قوانین تولید تهی را حذف می کنیم.

$$S \rightarrow Aa | b$$

$$A \rightarrow Ac | Sd | \epsilon$$

- متغیرها را به صورت  $S, A$  مرتب می کنیم. سپس قوانین زیر را تولید می کنیم.

$$S \rightarrow Aa | b$$

$$A \rightarrow Ac | Aad | bd | \epsilon$$

- با حذف بازگشت های چپ بลา واسطه گرامر زیر را به دست می آوریم.

$$S \rightarrow Aa | b$$

$$A \rightarrow bdA' | A'$$

$$A' \rightarrow cA' | adA' | \epsilon$$

## حذف بازگشت چپ

- گرامر زیر بازگشته چپ است.

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

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

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

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

## فاکتورگیری چپ

- فاکتورگیری چپ روشی است برای تبدیل کردن یک گرامر به گرامر که برای تجزیه کننده بالا به پایین پیش‌بینی‌کننده مناسب باشد.
- وقتی برای جایگزین کردن یک متغیر با بدنه قانون در فرایند اشتقاء 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.

```
stmt → if expr then stmt else stmt  
      | if expr then stmt
```

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

## فاکتورگیری چپ

- در حالت کلی اگر دو قانون  $\alpha\beta_1 \mid \alpha\beta_2 \rightarrow A$  را داشته باشیم و ورودی  $\alpha$  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 :

$$\begin{aligned}A &\rightarrow \alpha A' \\A' &\rightarrow \beta_1 \mid \beta_2\end{aligned}$$

## 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  $\alpha$  بین دو یا چند انتخاب را پیدا می‌کنیم. اگر  $\epsilon \neq \alpha$  آنگاه قوانین  $\gamma | \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n \rightarrow A$  را با قوانین زیر جایگزین می‌کنیم. قوانین  $\gamma$  قوانینی هستند که پیشوند آنها  $\alpha$  نیست.

$$\begin{aligned}A &\rightarrow \alpha A' | \gamma \\A' &\rightarrow \beta_1 | \beta_2 | \dots | \beta_n\end{aligned}$$

- 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 نماینده if و then و else هستند.

$$\begin{aligned} S &\rightarrow i \text{ } E \text{ } t \text{ } S \mid i \text{ } E \text{ } t \text{ } S \text{ } e \text{ } S \mid a \\ E &\rightarrow b \end{aligned}$$

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

$$\begin{aligned} S &\rightarrow i \text{ } E \text{ } t \text{ } S \text{ } S' \mid a \\ S' &\rightarrow e \text{ } S \mid \epsilon \\ E &\rightarrow b \end{aligned}$$

- 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  $\{w | w \in (a|b)^*\}$  مستقل از متن نیست.

## ساختارهای غیرمستقل از متن

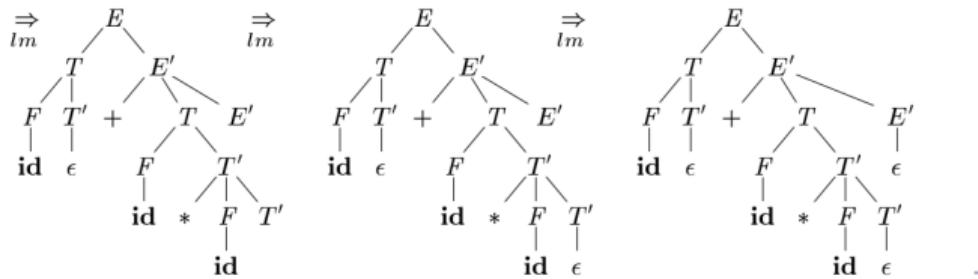
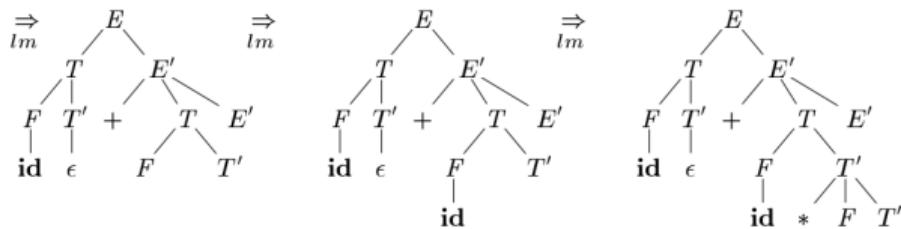
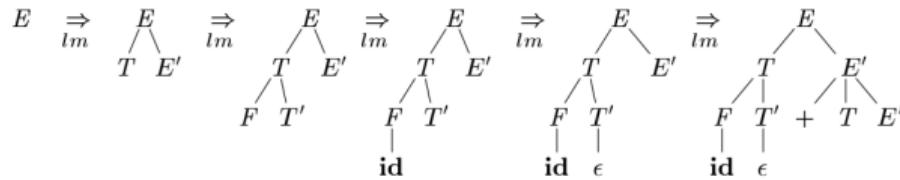
- در نتیجه نیاز به روش‌های دیگر برای تحلیل معنای برنامه‌ها داریم.
- یک مثال دیگر از ساختارهایی از زبان که مستقل از متن نیستند، به شرح زیر است. در زبان‌های برنامه‌نویسی نیاز است که تعداد آرگومان‌های ارسال شده به یک تابع برابر با تعداد پارامترهای تعریف شده در تابع باشد. فرض کنید تعریف دو تابع با  $n$ 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

## تجزیه بالا به پایین

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

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

# 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<sup>۱</sup> صحبت می‌کنیم که در آن برای پیدا کردن قانون مناسب در فرایند تجزیه از پسگرد<sup>۲</sup> استفاده می‌شود.
- سپس در مورد یک حالت خاص تجزیه کننده کاهشی بازگشتی به نام تجزیه کننده پیش‌بینی کننده<sup>۳</sup> صحبت می‌کنیم که در آن به پسگرد نیازی نیست.

---

<sup>۱</sup> recursive-descent parser

<sup>۲</sup> backtrack

<sup>۳</sup> 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.
- گرامرهايی که با بررسی  $k$  نماد در ورودی می‌توانيم برای آنها تجزیه کننده پیش بینی کننده بسازیم، گرامرهاي  $LL(k)$  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 همه رشته ورودی خوانده شد متوقف می‌شود.

# تجزیه کننده کاہشی بازگشتی

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

```
void A() {  
    1) Choose an  $A$ -production,  $A \rightarrow X_1 X_2 \cdots X_k$ ;  
    2) for (  $i = 1$  to  $k$  ) {  
        3)     if (  $X_i$  is a nonterminal )  
            4)         call procedure  $X_i()$ ;  
        5)     else if (  $X_i$  equals the current input symbol  $a$  )  
            6)         advance the input to the next symbol;  
        7)     else /* an error has occurred */;  
    }  
}
```

## تجزیه کننده کاھشی بازگشتی

- یک تجزیه کننده کاھشی بازگشتی با استفاده از یک الگوریتم پسگرد رشته ورودی را تجزیه می کند، اما برای تجزیه زبان های برنامه نویسی معمولاً نیازی به پسگرد نیست.
- برای اینکه در تجزیه بالا به پایین از پسگرد استفاده کنیم، در خط (۱) برنامه قبل باید همه انتخاب های موجود برای جایگزینی متغیر A را امتحان کنیم. همچنین در خط (۷) در صورتی که به بن بست برخورد کردیم پیام خطأ صادر نمی کنیم بلکه پسگرد انجام می شود.

## تجزیه کننده کاھشی بازگشتی

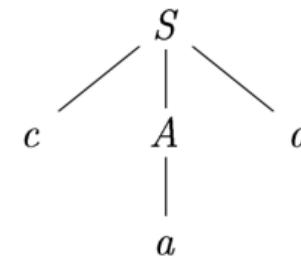
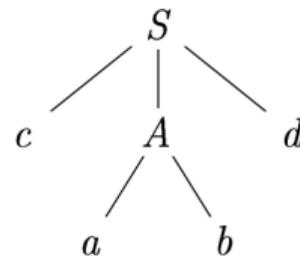
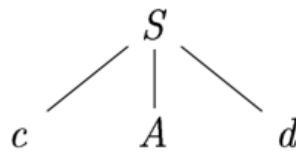
- گرامر زیر را در نظر بگیرید.

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab \mid a \end{aligned}$$

- 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 کاھشی بازگشتی

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



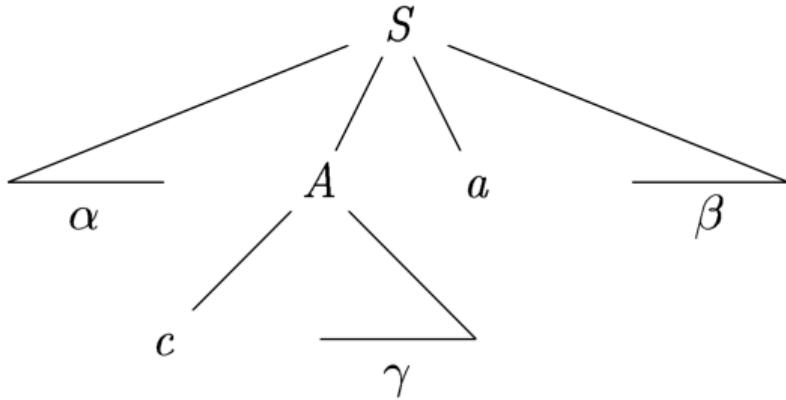
- اگر یک گرامر بازگشت چپ داشته باشد، تجزیه کننده کاھشی بازگشتی وارد حلقه بی پایان می شود.

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



- تابع First در تجزیه پیش‌بینی کننده استفاده می‌شود. فرض کنید دو قانون  $A \rightarrow \alpha|\beta$  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 حداقل در یکی از مجموعه‌های First( $\alpha$ ) یا First( $\beta$ ) باشد.

# توابع First و Follow

- به ازای متغیر  $A$  تابع  $\text{Follow}(A)$  مجموعه ترمینال‌های  $a$  است که مستقیماً در سمت راست متغیر  $A$  در یک صورت جمله‌ای در یک فرایند اشتقاق قرار می‌گیرند.
- به عبارت دیگر  $\text{Follow}(A)$  مجموعه ترمینال‌های  $a$  است که برای آنها اشتقاق  $S \xrightarrow{*} \alpha A a \beta \Rightarrow \alpha A a \beta$  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  $\text{Follow}(A)$  قرار می‌گیرد. نماد  $\$$  به معنای پایان رشته است و فرض می‌شود که این نماد در الفبا وجود ندارد.

# توابع First و Follow

- برای محاسبه  $\text{First}(X)$  برای نماد  $X$  قوانین زیر را اعمال می‌کنیم تا جایی که هیچ ترمینالی (یا رشته  $\epsilon$ ) نتواند به مجموعه  $\text{First}(X)$  اضافه شود.
۱. اگر  $X$  یک ترمینال است آنگاه  $\{\} = \text{First}(X)$ .
  ۲. اگر  $X$  یک متغیر است و  $X \rightarrow Y_1Y_2 \dots Y_k$  یک قانون تولید است به ازای  $k \geq 1$  آنگاه  $a$  در  $\text{First}(X)$  قرار می‌گیرد اگر به ازای یک  $a$  دلخواه، ترمینال  $a$  در  $\text{First}(Y_i)$  باشد و  $\epsilon$  در همه مجموعه‌های  $\text{First}(Y_1), \dots, \text{First}(Y_{i-1})$  باشد. در اینصورت خواهیم داشت  $\epsilon \xrightarrow{*} Y_1 \dots Y_{i-1}$ . اگر به ازای همه  $1 \leq j \leq k$ ، رشته  $\epsilon$  در  $\text{First}(Y_j)$  باشد آنگاه  $\epsilon$  را به  $\text{First}(X)$  اضافه می‌کنیم. برای مثال، هر نمادی در  $\text{First}(X)$  است در  $\text{First}(Y_1)$  نیز وجود دارد. اگر  $Y_1$  رشته تهی  $\epsilon$  را مشتق نمی‌کند آنگاه نماد دیگری به  $\text{First}(X)$  اضافه نمی‌کنیم، اما اگر  $\epsilon \xrightarrow{*} Y_1$  آنگاه  $\text{First}(Y_2)$  را نیز به  $\text{First}(X)$  می‌افزاییم و به همین ترتیب الى آخر.
  ۳. اگر  $\epsilon \rightarrow X$  یک قانون تولید باشد، آنگاه  $\epsilon$  را به  $\text{First}(X)$  می‌افزاییم.

# توابع First و Follow

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

# توابع First و Follow

- برای محاسبه  $\text{Follow}(A)$  به ازای همه متغیرهای  $A$  قوانین زیر را اعمال می‌کنیم تا وقتی که هیچ نمادی نتواند به مجموعه  $\text{Follow}(A)$  اضافه شود.
۱. اگر  $S$  متغیر آغازین باشد، نماد  $\$$  را در  $\text{Follow}(S)$  اضافه می‌کنیم.
  ۲. اگر قانون  $A \rightarrow \alpha B \beta$  وجود داشته باشد، آنگاه همه نمادهای مجموعه  $\text{First}(\beta)$  به جز رشته تهی  $\epsilon$  را به  $\text{Follow}(B)$  می‌افزاییم.
  ۳. اگر قانون  $A \rightarrow \alpha B \beta$  یا قانون  $A \rightarrow \alpha B$  وجود داشته باشد جایی که  $(\beta)$  حاوی  $\epsilon$  باشد، آنگاه هر نمادی در  $\text{Follow}(B)$  در  $\text{Follow}(A)$  نیز قرار می‌گیرد.

# توابع First و Follow

- برای مثال گرامر زیر را در نظر بگیرید.

$$S \rightarrow A^m \mid A^n \mid kA \mid B^p$$

$$A \rightarrow qrB \mid s$$

$$B \rightarrow t$$

- با محاسبه توابع First و Follow خواهیم داشت:

$$\text{First}(S) = \{ k, q, s, t \}$$

$$\text{Follow}(S) = \{ \$ \}$$

$$\text{First}(A) = \{ q, s \}$$

$$\text{Follow}(A) = \{ \$, m, n \}$$

$$\text{First}(B) = \{ t \}$$

$$\text{Follow}(B) = \{ \$, m, n, p \}$$

- 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 :

$$\text{First}(F) = \text{First}(T) = \text{First}(E) = \{(, \text{id}\}$$

$$\text{First}(E') = \{+, \epsilon\}$$

$$\text{First}(T') = \{*, \epsilon\}$$

$$\text{Follow}(E) = \text{Follow}(E') = \{(), \$\}$$

$$\text{Follow}(T) = \text{Follow}(T') = \{+, (), \$\}$$

$$\text{Follow}(F) = \{+, *, (), \$\}$$

# گرامرهای (1) LL

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

---

<sup>1</sup> Left to right

<sup>2</sup> Leftmost derivation

<sup>3</sup> one input symbol of lookahead

# گرامرهای (۱) LL(۱)

- گرامر مستقل از متن  $G$  یک گرامر (۱) LL است اگر و تنها اگر هنگامی که دو قانون مجزای  $\alpha \rightarrow \alpha|\beta$  وجود داشته باشند، شرط‌های زیر برقرار باشد.
۱. هیچ ترمینال  $a$  وجود ندارد به طوری که هردوی  $\alpha$  و  $\beta$  رشته‌ای مشتق کنند، که هر دو با  $a$  آغاز شود. همچنین حداکثر یکی از صورت‌های جمله‌ای  $\alpha$  و  $\beta$  می‌توانند رشته تهی تولید کنند.
  ۲. اگر  $\epsilon \xrightarrow{*} \beta$  آنگاه  $\alpha$  هیچ رشته‌ای تولید نمی‌کند که با یک ترمینال در  $\text{Follow}(A)$  آغاز شود. به طور مشابه اگر  $\epsilon \xrightarrow{*} \alpha$  آنگاه  $\beta$  هیچ رشته‌ای تولید نمی‌کند که با یک ترمینال در  $\text{Follow}(A)$  آغاز شود.  
فرض کنید این شرایط برقرار نباشد و داشته باشیم  $a \xrightarrow{*} S$  و همچنین  $Aa \xrightarrow{*} \beta a$ . در اینصورت نمی‌توانیم تصمیم بگیریم با مشاهده توکن  $a$  در ورودی در فرایند اشتقاء برای متغیر  $A$  کدامیک از قوانین  $\alpha \rightarrow A$  یا  $\beta \rightarrow A$  را انتخاب کنیم.

- شرط اول بدین معنی است که  $\text{First}(\alpha)$  و  $\text{First}(\beta)$  دو مجموعه مجزا هستند.
- شرط دوم بدین معنی است که اگر  $\epsilon$  در  $\text{First}(\beta)$  وجود داشت، آنگاه  $\text{First}(\alpha)$  و  $\text{Follow}(A)$  دو مجموعه مجزا هستند و به طور مشابه اگر  $\epsilon$  در  $\text{First}(\alpha)$  وجود داشت،  $\text{First}(\beta)$  و  $\text{Follow}(A)$  دو مجموعه مجزا هستند.

- تجزيه کننده‌های پيش‌بياني کننده می‌توانند برای گرامرهاي (1) LL استفاده شوند زيرا انتخاب درست قانوني
  - که می‌تواند در هرگام برای تجزيه به کار رود تنها با بررسی نماد بعدی در ورودی امكان پذير است.
  - برای مثال در گرامر زير تنها با خواندن يکی از نمادهای if یا while یا { می‌توانيم تصميم بگيريم كدام قانون را انتخاب کنيم.
- 

```
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  $\alpha \rightarrow A$  انتخاب می‌شود اگر نماد بعدی  $a$  در  $First(\alpha)$  باشد. در صورتی که  $\epsilon = \epsilon^*$  یا  $\alpha \Rightarrow^* \epsilon$  آنگاه  $\alpha \rightarrow A$  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  $\alpha \rightarrow A$  از گرامر به صورت زیر عمل می‌کنیم.
- ۱. به ازای هریک از ترمینال‌های  $a$  در  $First(\alpha) \rightarrow A$  قانون  $First(A, a) \rightarrow A$  را به  $M[A, a]$  اضافه می‌کنیم.
- ۲. اگر  $\epsilon$  در  $First(\alpha)$  باشد، آنگاه به ازای هریک از ترمینال‌های  $a$  در  $Follow(A) \rightarrow \alpha$  قانون  $Follow(A) \rightarrow \alpha$  را به  $M[A, a]$  اضافه می‌کنیم. اگر  $\epsilon$  در  $First(\alpha)$  باشد و  $\$$  در  $Follow(A) \rightarrow \alpha$  باشد آنگاه  $M[A, \$]$  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، خطاهای را با خانه‌های خالی در جدول نمایش می‌دهیم.

# الگوریتم تجزیه کننده پیش‌بینی کننده

- برای گرامر زیر جدول تجزیه زیر تولید می‌شود.

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

NON - TERMINAL	INPUT SYMBOL					
	<b>id</b>	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

# الگوریتم تجزیه کننده پیش‌بینی کننده

- قانون  $E \rightarrow TE'$  را در نظر بگیرید. از آنجایی که  $\text{First}(TE') = \{(, \text{id}\}$  این قانون به  $M[E, ()]$  و  $M[E, \text{id}]$  افزوده شده است.
- قانون  $E' \rightarrow +TE'$  را در نظر بگیرید. از آنجایی که  $\text{First}(+TE') = \{+\}$  این قانون به  $M[E', +]$  افزوده شده است زیرا  $E' \rightarrow +TE'$ .
- از آنجایی که  $\text{Follow}(E') = \{\}, \$\}$  این قانون به  $E' \rightarrow \epsilon$  افزوده شده است.

NON - TERMINAL	INPUT SYMBOL					
	<b>id</b>	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

## الگوریتم تجزیه کننده پیش‌بینی کننده

- الگوریتمی که شرح داده شد می‌تواند برروی هر گرامر  $G$  اعمال شود و یک جدول تجزیه  $M$  بسازد. برای هر گرامر  $LL(1)$  یک جدول تجزیه وجود دارد که هر خانه 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  $LL(1)$  تبدیل کرد. اما برای برخی از گرامرها معادل  $LL(1)$  وجود ندارد.

الگوریتم تجزیه کننده پیش‌بینی کننده

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

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

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 [S', e] قانون  $S' \rightarrow eS$  و  $S' \rightarrow \epsilon$  قرار گرفته است.

- دلیل این امر این است که گرامر مبهم است.

## تجزیه کننده پیش‌بینی کننده غیر بازگشتی

- یک تجزیه کننده پیش‌بینی کننده غیر بازگشتی<sup>۱</sup> با نگهداری یک پشته به صورت صریح به جای استفاده از پشته فراخوانی ساخته می‌شود.
- این تجزیه کننده اشتقاد چپ را شبیه‌سازی می‌کند.
- اگر  $w$  رشته ورودی باشد که بر گرامر تطبیق داده شده باشد، آنگاه پشته یک دنباله از نمادهای  $\alpha$  از گرامر را نگهداری می‌کند به طوری که  $w\alpha \xrightarrow{*} S_{lm}$ .
- 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.

---

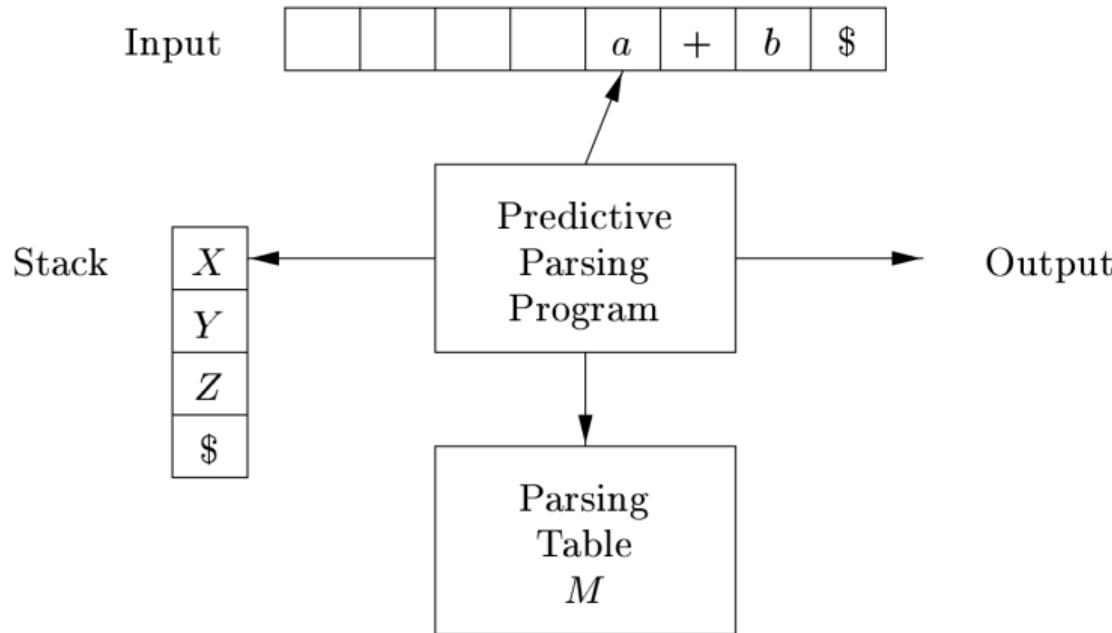
<sup>۱</sup> nonrecursive predictive parser

## تجزیه کننده پیش‌بینی کننده غیر بازگشتی

- بافر ورودی رشته ورودی را در برمی‌گیرد 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 غیر اینصورت نماد  $X$  و نماد ورودی  $a$  باید تطبیق داده شوند.

# تجزیه کننده پیش‌بینی کننده غیر بازگشتی

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



## تجزیه کننده پیش‌بینی کننده غیر بازگشتی

- 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 غیر اینصورت پیام خطا صادر می‌کند.
- در ابتدا تجزیه کننده در پیکربندی  $^1 w\$$  قرار دارد و نماد آغازین  $S$  در پشته بروی نماد انتهای پشته یعنی  $\$$  قرار گرفته می‌شود.

---

<sup>1</sup> configuration

## تجزیه کننده پیش‌بینی کننده غیر بازگشتی

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 ( $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$ ) {
        output the production  $X \rightarrow Y_1 Y_2 \dots Y_k$ ;
        pop the stack;
        push  $Y_k, Y_{k-1}, \dots, Y_1$  onto the stack, with  $Y_1$  on top;
    }
    let  $X$  be the top stack symbol;
}
```

## تجزیه کننده پیش‌بینی کننده غیر بازگشتی

- گرامر زیر را در نظر بگیرید.

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

# تجزیه کننده پیش‌بینی کننده غیر بازگشتی

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

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

## تجزیه کننده پیش‌بینی کننده غیر بازگشتی

با دریافت ورودی  $\text{id} * \text{id}$  تجزیه کننده پیش‌بینی کننده غیر بازگشتی یک فرایند اشتقاق چپ به صورت زیر تولید می‌کند.

$$E \xrightarrow{lm} TE' \xrightarrow{lm} FT'E' \xrightarrow{lm} idT'E' \xrightarrow{lm} idE' \xrightarrow{lm} id + TE' \xrightarrow{lm} \dots$$

# تجزیه کننده پیش‌بینی کننده غیر بازگشتی

- این فرایند اشتقاق به صورت زیر تولید می‌شود.

MATCHED	STACK	INPUT	ACTION
	$E\$$	$id + id * id\$$	
	$TE' \$$	$id + id * id\$$	output $E \rightarrow TE'$
	$FT'E' \$$	$id + id * id\$$	output $T \rightarrow FT'$
	$id T'E' \$$	$id + id * id\$$	output $F \rightarrow id$
$id$	$T'E' \$$	$+ id * id\$$	match $id$
$id$	$E' \$$	$+ id * id\$$	output $T' \rightarrow \epsilon$
$id$	$+ TE' \$$	$+ id * id\$$	output $E' \rightarrow + TE'$
$id +$	$TE' \$$	$id * id\$$	match $+$
$id +$	$FT'E' \$$	$id * id\$$	output $T \rightarrow FT'$
$id +$	$id T'E' \$$	$id * id\$$	output $F \rightarrow id$
$id + id$	$T'E' \$$	$* id\$$	match $id$
$id + id$	$* FT'E' \$$	$* id\$$	output $T' \rightarrow * FT'$
$id + id *$	$FT'E' \$$	$id\$$	match $*$
$id + id *$	$id T'E' \$$	$id\$$	output $F \rightarrow id$
$id + id * id$	$T'E' \$$	$\$$	match $id$
$id + id * id$	$E' \$$	$\$$	output $T' \rightarrow \epsilon$
$id + id * id$	$\$$	$\$$	output $E' \rightarrow \epsilon$

## تجزیه کننده پیش‌بینی کننده غیر بازگشتی

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

## بازیابی خطای در تجزیه کننده پیش‌بینی کننده

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

## بازیابی خطا با توکن همگام‌کننده

- بازیابی خطا با توکن همگام‌کننده<sup>۱</sup> بر این پایه است که در هنگام رخداد خطا از نمادهای ورودی چشمپوشی شود تا جایی که یک توکن همگام‌کننده<sup>۲</sup> پیدا شود.
- مجموعه توکن‌های همگام‌کننده باید به نحوی انتخاب شود که تجزیه‌کننده بتواند به سرعت خطا را بازیابی کند.

---

<sup>۱</sup> panic mode error recovery

<sup>۲</sup> synchronizing token

## بازیابی خطای با توکن همگام کننده

- از قوانین زیر می‌توان در بازیابی خطای با توکن همگام کننده استفاده کرد.
- ۱. همه نمادها در  $\text{Follow}(A)$  را در مجموعه همگام کننده متغیر  $A$  قرار می‌دهیم. اگر از همه توکن‌ها چشم‌پوشی کنیم تا یکی از اعضای  $\text{Follow}(A)$  مشاهده شود و  $A$  از پشته خارج شود، به احتمال زیاد تجزیه می‌تواند ادامه پیدا کند.
- برای مثال فرض کنید قوانین  $E' \rightarrow + id\ E' | \epsilon$  و  $E \rightarrow id\ E' | S$  و  $S \rightarrow E; S | \epsilon$  در یک گرامر وجود داشته باشد. برای تجزیه جمله  $E'; S\$$  وقتی در پشته نمادهای  $E'; S\$$  قرار داشته باشند و به عبارت  $(id + id; id + id)$  بخورد کنیم، از کاراکترها چشم‌پوشی می‌شود تا اینکه به یک کاراکتر نقطه ویرگول بخورد کنیم، زیرا  $\{ ; \}$   $\text{Follow}(E')$  است. سپس  $E'$  از پشته خارج می‌شود و نقطه ویرگول در پشته تطبیق داده می‌شود و تجزیه با عبارت  $id + id$  ادامه پیدا می‌کند.

## بازیابی خطاب با توکن همگام‌کننده

- تنها اعضای 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 الى آخر. می‌توانیم نمادهایی را که متغیرهای سلسله مراتب 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  $E \rightarrow id\ E' | int\ E; S | \epsilon$  و  $E' \rightarrow +\ id\ E' | int\ E$  در یک گرامر وجود داشته باشد. از آنجایی که E می‌تواند با کلمه int آغاز شود، آن را به مجموعه همگام‌کننده E' می‌افزاییم. اگر در پشته نمادهای E'; S\$ قرار داشته باشند و به عبارت int id; (id + id) برخورد کنیم، از کarakترها چشم‌پوشی می‌شود تا اینکه به یک کاراکتر نقطه ویرگول یا توکن int برخورد کنیم، زیرا  $. \ synch(E') = \{ ; , int \}$

# بازیابی خطای با توکن همگام کننده

۲. در صورتی که در ورودی با خطای روبرو شدیم و در پشتی متغیر  $A$  قرار داشته باشد، می‌توانیم از توکن‌ها چشم‌پوشی کنیم تا به یک کاراکتر در  $\text{First}(A)$  بخورد کنیم. آنگاه می‌توانیم تجزیه را با توجه به متغیر  $A$  ادامه دهیم، اگر یک نماد در  $\text{First}(A)$  در ورودی ظاهر شود.

- برای مثال فرض کنید قوانین  $E' \rightarrow + \text{id} E' | \epsilon$  و  $E \rightarrow \text{id} E; S | \epsilon$  در یک گرامر وجود داشته باشد. در تجزیه  $(+ \text{id}; S\$)$  وقتی در پشتی نمادهای  $E'$  و  $S\$$  قرار داشته باشند و به عبارت  $((+ \text{id}; S\$))$  بخورد کنیم، از کاراکترهای  $( )$  چشم‌پوشی می‌شود و یک پیام خطای صادر می‌شود تا اینکه به یک کاراکتر  $+$  بخورد کنیم، زیرا  $\{+\}$ .  $\text{First}(E') = \{+\}$

## بازیابی خطای با توکن همگام کننده

۳. اگر یک ترمینال بر روی پشته باشد که نتواند تطبیق داده شود، می‌توان ترمینال را از روی پشته برداشته، خطایی صادر کرد مبنی بر اینکه ترمینال توسط کامپایلر اضافه شده است و عملیات تجزیه را ادامه داد.
- برای مثال فرض کنید قوانین  $E \rightarrow E; S \mid \epsilon$  و  $E' \rightarrow + id E'$  در یک گرامر وجود داشته باشد. در تجزیه  $id id$ ; وقتی در پشته نمادهای  $id E; S\$$  قرار داشته باشند و به عبارت  $id$  برخورد کنیم، توکن  $+$  از پشته برداشته می‌شود و یک پیام خطای صادر می‌شود، مبنی بر اینکه توکن  $+$  در ورودی فراموش شده است، و تجزیه ادامه پیدا می‌کند.

# بازیابی خطاب با توکن همگام کننده

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

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

NON - TERMINAL	INPUT SYMBOL					
	<b>id</b>	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$		
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

# بازیابی خطاب با توکن همگام‌کننده

در جدول زیر واژه  $\text{synch}$  در خانه  $M[A, a]$  قرار گرفته است، اگر  $a \in \text{Follow}(A)$  باشد.

NON - TERMINAL	INPUT SYMBOL					
	<b>id</b>	+	*	(	)	\$
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$	synch	synch	$F \rightarrow (E)$	synch	synch

## بازیابی خطاب با توکن همگام‌کننده

- از این جدول به صورت زیر استفاده می‌شود.
- (قانون ۱) اگر تجزیه کننده به کلمه `synch` برخورد کرد، متغیر از روی پشته برداشته می‌شود و پیام خطاب صادر می‌شود تا تجزیه بتواند ادامه پیدا کند.
- (قانون ۲) اگر تجزیه کننده به سلول `M[A, a]` رسید که خالی بود آنگاه پیام خطاب صادر شده، از `a` چشم‌پوشی می‌شود.
- (قانون ۳) اگر یک توکن از روی پشته بر نماد ورودی تطبیق داده نشود، آنگاه توکن از پشته برداشته می‌شود و پیام خطاب صادر می‌شود مبنی بر اینکه توکن مورد نظر اضافه شده است.

# بازیابی خطاب با توکن همگام کننده

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

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

## بازیابی خطای با توکن همگام‌کننده

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

# بازیابی خطای با جایگزینی توکن‌ها

- بازیابی خطای با جایگزینی توکن‌ها<sup>۱</sup> بدین صورت پیاده‌سازی می‌شود که به جای سلول‌های خالی در جدول تجزیه، توابعی قرار می‌گیرند که بازیابی خطای را انجام می‌دهند. این توابع می‌توانند نمادهایی را تغییر دهند یا اضافه کنند و یا حذف کنند و پیام خطای مناسب صادر کنند. همچنین آین 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.

---

<sup>۱</sup> phrase-level error recovery

## تجزیه پایین به بالا

- یک تجزیه کننده پایین به بالا<sup>۱</sup> درخت تجزیه برای یک ورودی را از برگ‌ها (پایین) به سمت ریشه (بالا) می‌سازد.

---

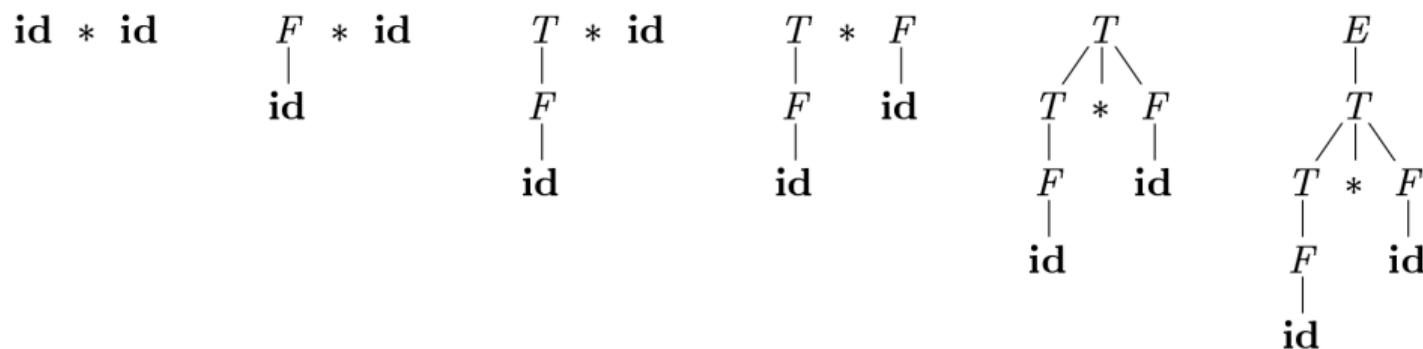
<sup>۱</sup> bottom-up parser

## تجزیه پایین به بالا

- فرض کنید می خواهیم رشته  $\text{id} * \text{id}$  را با استفاده از یک تجزیه کننده پایین به بالا برای گرامر زیر تجزیه کنیم.

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

- 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<sup>۱</sup> معرفی می‌کنیم.
- یکی از دسته‌های مهم گرامرها که برای آنها تجزیه کننده انتقال کاهش می‌تواند ساخته شود، دسته گرامرها LR نامیده می‌شود.

---

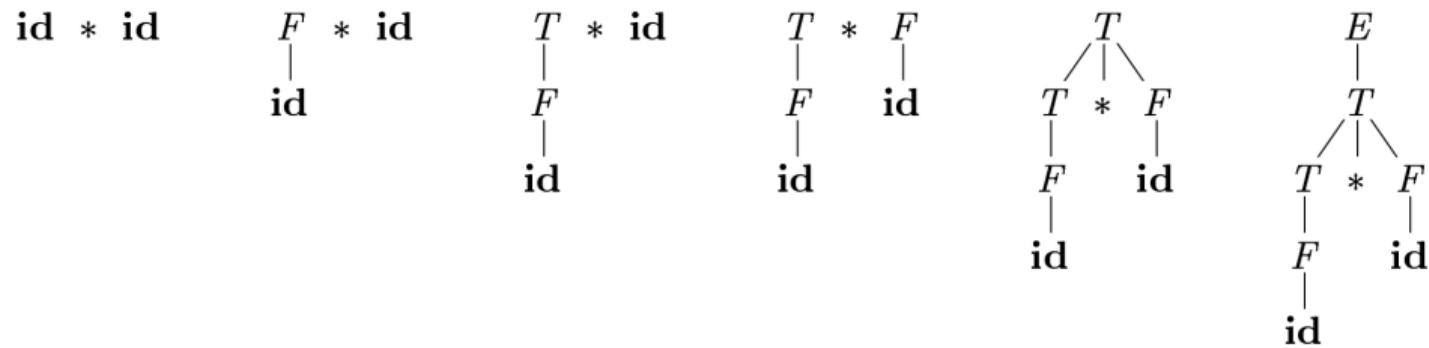
<sup>1</sup> shift-reduce parsing

یک تجزیه کننده پایین به بالا با دریافت یک رشته ورودی آن را به متغیر آغازین کاهش می‌دهد. در هرگام کاهش<sup>۱</sup>، یک زیررشته از ورودی بر بدن<sup>ه</sup> 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.

---

<sup>۱</sup> reduction

- در شکل زیر  $id * id * F$  کاهاش پیدا می‌کند و سپس به ترتیب به  $T * id$ ،  $T * F$ ،  $T$  و در نهایت رشته ورودی به  $E$  کاهاش پیدا می‌کند.



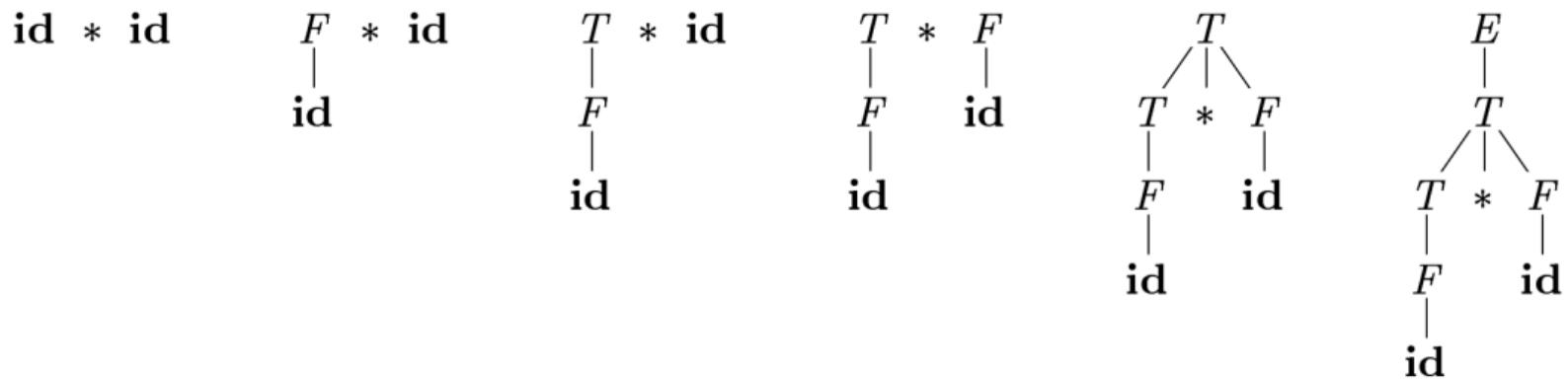
- در گام اول برای کاهاش از قانون  $id \rightarrow F$  استفاده می‌شود. در برخی از گام‌ها چند انتخاب برای کاهاش وجود دارد که تجزیه کنند باید تصمیم بگیرید از کدام قانون و کدام زیر رشته برای کاهاش استفاده کند.

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 بدنه یک قانون منطبق و با متغیر متعلق به آن قانون جایگزین می‌شود. بنابراین تجزیه کننده پایین به بالا یک اشتقاق به صورت معکوس می‌سازد.

- فرایند اشتقاق راست زیر را در نظر بگیرید.

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$$

- در شکل زیر این فرایند اشتقاق از آخر به اول انجام می‌شود.



- در تجزیه پایین به بالا ورودی از چپ به راست خوانده می‌شود و یک اشتقاق راست<sup>۱</sup> به صورت معکوس تولید می‌شود.
- یک هندل<sup>۲</sup>، زیر رشته‌ای است که بر بدنی 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.

---

<sup>۱</sup> rightmost derivation

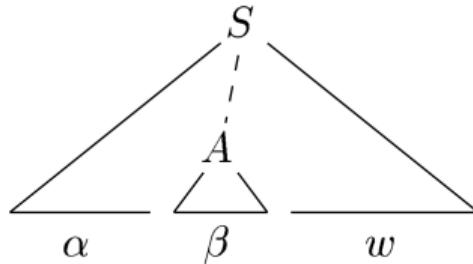
<sup>۲</sup> handle

- برای مثال در جدول زیر هندل در هر گام از فرایند کاهش مشخص شده است. برای خوانایی بیشتر و تمیز دادن توکن‌های  $\text{id}$  ، از اندیس استفاده شده است.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\text{id}_1 * \text{id}_2$	$\text{id}_1$	$F \rightarrow \text{id}$
$F * \text{id}_2$	$F$	$T \rightarrow F$
$T * \text{id}_2$	$\text{id}_2$	$F \rightarrow \text{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
$T$	$T$	$E \rightarrow T$

- در این فرایند کاهش در گام سوم گرچه  $T$  در  $\text{id}_2 * T$  می‌تواند به عنوان هندل انتخاب شود چرا که قانون  $T \rightarrow E$  وجود دارد، اما  $T$  به عنوان هندل انتخاب نمی‌شود چرا که در این صورت متغیر آغازین  $E$  به دست نمی‌آید و رشته تجزیه نمی‌شود. همچنین در گام چهارم می‌توانستیم  $F$  را به  $T$  کاهش دهیم، اما در آن صورت رشته تجزیه نمی‌شد.

- اگر داشته باشیم  $S \xrightarrow{r_m} \alpha Aw \xrightarrow{r_m} \alpha\beta w$  در فرایند کاهش است.



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

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

$$S = \gamma_0 \xrightarrow{r_m} \gamma_1 \xrightarrow{r_m} \cdots \xrightarrow{r_m} \gamma_{n-1} \xrightarrow{r_m} \gamma_n = w$$

برای ساختن این اشتقاد به صورت معکوس، هندل  $\gamma_n$  در  $\beta_n$  پیدا می‌شود و با  $A_n$  با استفاده از قانون  $A_n \rightarrow \beta_n$  جایگزین می‌شود تا صورت جمله  $\gamma_{n-1}$  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 این صورت رشته تجزیه شده است و متعلق به گرامر است.

- دقت کنید در فرایند اشتاقاق راست ابتدا قسمت راست صورت‌های جمله‌ای تجزیه می‌شود، یعنی در یک فرایند اشتاقاق داریم  $\alpha Aw \Rightarrow \alpha\beta w$  که  $\beta$  جایی است.
- بنابراین اگر بخواهیم صورت جمله‌ای  $\alpha\beta w$  را کاهش بدھیم، به  $\alpha Aw$  می‌رسیم.
- در هیچ فرایند اشتاقاق راست، اشتاقاق  $\alpha A\gamma Bw \Rightarrow \alpha\beta\gamma Bw$  ممکن نیست.
- بنابراین هیچ فرایند کاهشی وجود ندارد که یک صورت جمله‌ای به صورت  $\alpha A\gamma Bw$  را به  $\alpha\beta\gamma Bw$  کاهش دهد.

## تجزیه انتقال کاهش

- تجزیه انتقال کاهش 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 بروی 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 انتقال کاهش

- 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  $\beta$  برروی پشته برای کاهش یافت شود. سپس  $\beta$  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 خطأ روبرو شود و یا در پشته متغیر آغازین قرار بگیرد و ورودی به پایان برسد. در اینصورت وضعیت پشته و رشته ورودی به صورت زیر است و رشته ورودی به درستی تجزیه شده است.

STACK

$\$ S$

INPUT

$\$$

## جزیه انتقال کاهش

- شکل زیر گام‌های یک تجزیه کننده انتقال کاهش را برای تجزیه رشته  $\text{id} * \text{id}$  نشان می‌دهد.

STACK	INPUT	ACTION
\$	$\text{id}_1 * \text{id}_2 $$	shift
\$ <b><math>\text{id}_1</math></b>	$* \text{id}_2 $$	reduce by $F \rightarrow \text{id}$
\$ <b><math>F</math></b>	$* \text{id}_2 $$	reduce by $T \rightarrow F$
\$ <b><math>T</math></b>	$* \text{id}_2 $$	shift
\$ <b><math>T * </math></b>	<b><math>\text{id}_2 \$</math></b>	shift
\$ <b><math>T * \text{id}_2</math></b>	\$	reduce by $F \rightarrow \text{id}$
\$ <b><math>T * F</math></b>	\$	reduce by $T \rightarrow T * F$
\$ <b><math>T</math></b>	\$	reduce by $E \rightarrow T$
\$ <b><math>E</math></b>	\$	accept

# تجزیه انتقال کاهش

- در فرایند تجزیه انتقال کاهش چهار عملیات می‌تواند توسط تجزیه کننده اجرا شود.
  ۱. انتقال<sup>۱</sup> : یک نماد از ورودی به روی پشته انتقال پیدا می‌کند.
  ۲. کاهش<sup>۲</sup> : یک هندل که نماد سمت راست آن در 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 تولید کننده هندل بر روی پشته اضافه می‌شود. هندل همیشه بالای پشته قرار می‌گیرد نه در وسط آن.
  ۳. پذیرش<sup>۳</sup> : عملیات تجزیه به اتمام رسیده و رشتہ پذیرفته شده است.
  ۴. خطأ<sup>۴</sup> : یک خطأ نحوی تشخیص داده شده و یک تابع بازیابی خطأ فراخوانی می‌شود.

---

<sup>۱</sup> shift

<sup>۲</sup> reduce

<sup>۳</sup> accept

<sup>۴</sup> error

# تعارض در تجزیه انتقال کاهش

- برای برخی از گرامرها می‌توان از متن تجزیه انتقال کاهش نمی‌تواند استفاده شود. در چنین گرامرها می‌توان تجزیه کننده انتقال کاهش به یک پیکربندی می‌رسد که در آن تجزیه کننده نمی‌تواند تصمیم بگیرد عملیات انتقال انجام دهد و یا عملیات کاهش. به این شرایط تعارض انتقال کاهش<sup>۱</sup> گفته می‌شود. همچنین ممکن است تجزیه کننده نمی‌تواند تصمیم بگیرد از بین چند کاهش کدام یک را اعمال کند. به این شرایط تعارض کاهش کاهش<sup>۲</sup> گفته می‌شود.

---

<sup>1</sup> shift/reduce conflict

<sup>2</sup> reduce/reduce conflict

## عارض در تجزیه انتقال کاهش

- اگر یک تجزیه کننده انتقال کاهش با آگاهی از  $k$  نماد بعدی 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 می شود.
- گرامری که تجزیه کننده انتقال کاهش برای تجزیه جملات با آن تنها نیاز به آگاهی از یک نماد بعدی در ورودی داشته باشد، یک گرامر  $LR(1)$  نامیده می شود.
- حرف  $L$  بدین معنی است که ورودی از چپ به راست<sup>1</sup> خوانده می شود و حرف  $R$  بدین معنا است که یک اشتقاق راست به صورت معکوس<sup>2</sup> ایجاد می شود و  $k$  بدین معنی است که آگاهی از  $k$  نماد بعدی از ورودی برای تجزیه جمله کافی است.

---

<sup>1</sup> left-to-right

<sup>2</sup> rightmost derivation in reverse

## تعارض در تجزیه انتقال‌کاهش

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

```
stmt → if expr then stmt  
      | if expr then stmt else stmt  
      | other
```

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

STACK	INPUT
... if expr then stmt	else ... \$

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

## تعارض در تجزیه انتقال کاهش

- در برخی مواقع تجزیه کننده نمی‌تواند تصمیم بگیرد از بین چند هنل کدام یک را انتخاب کند و کدام قانون تولید را در فرایند 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

(1)	<i>stmt</i>	$\rightarrow$	<b>id</b> ( <i>parameter_list</i> )
(2)	<i>stmt</i>	$\rightarrow$	<i>expr</i> := <i>expr</i>
(3)	<i>parameter_list</i>	$\rightarrow$	<i>parameter_list</i> , <i>parameter</i>
(4)	<i>parameter_list</i>	$\rightarrow$	<i>parameter</i>
(5)	<i>parameter</i>	$\rightarrow$	<b>id</b>
(6)	<i>expr</i>	$\rightarrow$	<b>id</b> ( <i>expr_list</i> )
(7)	<i>expr</i>	$\rightarrow$	<b>id</b>
(8)	<i>expr_list</i>	$\rightarrow$	<i>expr_list</i> , <i>expr</i>
(9)	<i>expr_list</i>	$\rightarrow$	<i>expr</i>

## عارض در تجزیه انتقال کاهاش

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

STACK  
... **id** ( **id**

INPUT  
, **id** ) ...

- در اینجا دو قانون ۵ برای کاهاش **id** به پارامتر تابع و قانون ۷ برای کاهاش **id** به نام پارامتر آرایه می‌توانند استفاده شوند و یک ععارض کاهاش کاهاش به وجود می‌آید.

- یک راه حل این است که به جای **id** در قانون تولید ۱ از توکن **procid** استفاده شود. این راه حل تحلیل گر لغوی را پیچیده می‌کند، زیرا تحلیل گر باید با استفاده از جدول علائم تشخیص دهد یک شناسه نام تابع است یا نام آرایه.

- یک راه حل 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<sup>۱</sup> خوانده می‌شود و حرف R بدین معناست که تجزیه با استفاده از یک فرایند اشتراق راست معکوس<sup>۲</sup> انجام می‌شود و  $k$  به معنای تعداد نمادهای بعدی است که برای تصمیم‌گیری در فرایند تجزیه استفاده می‌شود.
- برای تجزیه زبان‌های برنامه‌نویسی معمولاً از تجزیه‌کننده LR(1) استفاده می‌شود.
- وقتی از تجزیه‌کننده LR صحبت می‌کنیم منظور تجزیه‌کننده LR(1) است.
- ابتدا در مورد یک تجزیه‌کننده LR ساده<sup>۳</sup> یا SLR صحبت می‌کنیم و سپس با روش‌های پیچیده‌تر تجزیه از جمله تجزیه‌کننده LR استاندارد<sup>۴</sup> یا CLR و تجزیه‌کننده LALR آشنا می‌شویم.

---

<sup>۱</sup> left-to-right

<sup>۲</sup> rightmost derivation in reverse

<sup>۳</sup> Simple LR (SLR)

<sup>۴</sup> Canonical LR (CLR)

## 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. تجزیه کننده‌های 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. تجزیه کننده LR روشی است غیربازگشتی برای پیاده‌سازی تجزیه انتقال‌کاهش و در عین حال به اندازه بقیه روش‌های تجزیه کاراست. گرامر مورد تجزیه نیاز به حذف بازگشت چپ ندارد و در برخی موقع می‌تواند مبهم نیز باشد.
  ۳. روش‌های کارایی برای بازیابی خطأ در تجزیه کننده LR وجود دارد.
  ۴. دسته گرامرهای LR ابر مجموعه دسته گرامرهای LL است، پس تجزیه کننده LR تعداد گرامرهای بیشتری را پوشش می‌دهد.
- تنها عیب تجزیه کننده LR این است که ساختن آن بسیار پیچیده است.

## ماشین $LR(0)$

- چگونه یک تجزیه‌کننده انتقال کاهش 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 $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

## ماشین $(^{\circ})$ LR

- یک تجزیه‌کننده LR تصمیم انتقال یا کاهش را با نگهداری تعدادی حالت انجام می‌دهد.
- هر یک از این حالت‌ها، مجموعه‌ای از آیتم‌ها<sup>۱</sup> را در بر می‌گیرند.
- یک آیتم  $(^{\circ})$ LR از گرامر G یک قانون تولید گرامر G است که تعدادی نقطه در بین نمادهای بدنه آن افزوده شده‌اند.
- بنابراین قانون  $XYZ \rightarrow A$  چهار آیتم به صورت زیر دارد :
  - $A \rightarrow \cdot XYZ$
  - $A \rightarrow X \cdot YZ$
  - $A \rightarrow XY \cdot Z$
  - $A \rightarrow XYZ \cdot$
- قانون تولید  $\epsilon \rightarrow A$  تنها یک آیتم به صورت  $\cdot \rightarrow A$  دارد.

---

<sup>1</sup> item

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

# ماشین $(^{\circ})LR$

- یک گروه از مجموعه آیتم‌های  $LR(0)$ <sup>۱</sup> یک گروه  $(^{\circ})LR$ <sup>۲</sup> استاندارد نامیده می‌شود که از آن برای ساختن یک ماشین متناهی قطعی برای تصمیم‌گیری در فرایند ترجمه استفاده می‌شود.
- این ماشین را ماشین  $(^{\circ})LR$ <sup>۳</sup> می‌نامیم.

---

<sup>۱</sup> collection of sets of  $LR(0)$  items

<sup>۲</sup> canonical  $LR(0)$  collection

<sup>۳</sup>  $LR(0)$  automaton

## ماشین $LR(0)$

- هر حالت از ماشین  $LR(0)$  نشان دهنده مجموعه‌ای از آیتم‌ها در گروه  $LR(0)$  استارندارد<sup>۱</sup> است.
- برای ساختن گروه  $LR(0)$  استارندارد برای یک گرامر، یک گرامر افزوده شده و دو تابع Closure و Goto را تعریف می‌کنیم.
- اگر  $G$  یک گرامر با نماد آغازین  $S$  باشد، آنگاه  $G'$  یک گرامر افزوده شده<sup>۲</sup> برای  $G$  است که نماد آغازین  $'S'$  و قانون  $S \rightarrow S'$  در آن افزوده شده است.

---

<sup>۱</sup> canonical  $LR(0)$  collection

<sup>۲</sup> augmented grammar

- اگر  $I$  مجموعه‌ای از آیتم‌ها بر روی گرامر  $G$  باشد، آنگاه  $\text{Closure}(I)$  مجموعه‌ای از آیتم‌ها از  $I$  است که با دو قانون زیر ساخته شده‌اند :

  ۱. هر آیتم در  $I$  در  $\text{Closure}(I)$  نیز اضافه می‌شود.
  ۲. اگر  $\alpha \cdot B\beta \in \text{Closure}(I)$  در  $A \rightarrow \gamma$  قانون تولید باشد، آنگاه آیتم  $\gamma \rightarrow B$  یک قانون تولید باشد، آنگاه آیتم  $\gamma \rightarrow B$  در صورتی که در  $\text{Closure}(I)$  وجود نداشته باشد، به آن اضافه می‌شود. این کار تکرار می‌شود تا جایی که هیچ آیتم دیگری را نتوان به  $\text{Closure}(I)$  اضافه کرد.

- به طور شهودی  $\alpha \cdot B\beta \rightarrow A$  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 می‌کنیم. پس اگر  $\gamma \rightarrow B$  یک قانون تولید باشد،  $\gamma \cdot B \rightarrow A$  در Closure(I) قرار می‌گیرد.

# Closure تابع

- مثال : گرامر زیر را در نظر بگیرید.

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \quad | \quad T \\ T & \rightarrow & T * F \quad | \quad F \\ F & \rightarrow & (E) \quad | \quad \text{id} \end{array}$$

- اگر  $I$  مجموعه‌ای از یک آیتم از  $\{[E'] \rightarrow \cdot E\}$  باشد، آنگاه  $\text{Closure}(I)$  مجموعه آیتم‌های  $I_0$  را شامل می‌شود.

$$\begin{array}{c} I_0 \\ \overline{E' \rightarrow \cdot E} \\ \overline{E \rightarrow \cdot E + T} \\ \overline{E \rightarrow \cdot T} \\ \overline{T \rightarrow \cdot T * F} \\ \overline{T \rightarrow \cdot F} \\ \overline{F \rightarrow \cdot (E)} \\ \overline{F \rightarrow \cdot \text{id}} \end{array}$$

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

- تابع Closure را می‌توان براساس الگوریتم زیر تولید کرد.

```

SetOfItems CLOSURE( $I$ ) {
     $J = I;$ 
    repeat
        for ( each item  $A \rightarrow \alpha \cdot B \beta$  in  $J$  )
            for ( each production  $B \rightarrow \gamma$  of  $G$  )
                if (  $B \rightarrow \cdot \gamma$  is not in  $J$  )
                    add  $B \rightarrow \cdot \gamma$  to  $J$ ;
    until no more items are added to  $J$  on one round;
    return  $J$ ;
}

```

- مجموعه‌های آیتم‌ها را به دو دسته تقسیم می‌کنیم.
- ۱. آیتم‌های هسته<sup>۱</sup> : آیتم شروع  $S \rightarrow S'$  و همه آیتم‌هایی که نقطه در سمت چپ بدن آنها نیست.
- ۲. آیتم‌های غیرهسته<sup>۲</sup> : همه آیتم‌هایی که نقطه در سمت چپ بدن آنهاست به جز  $S \rightarrow S'$ .
- هر مجموعه از آیتم‌ها تشکیل شده است از Closure بروی مجموعه آیتم‌های هسته.
- بنابراین برای صرفه‌جویی در حافظه می‌توانیم آیتم‌های غیر هسته را دور بریزیم زیرا این آیتم‌ها مجدداً می‌توانند از آیتم‌های هسته محاسبه شوند.

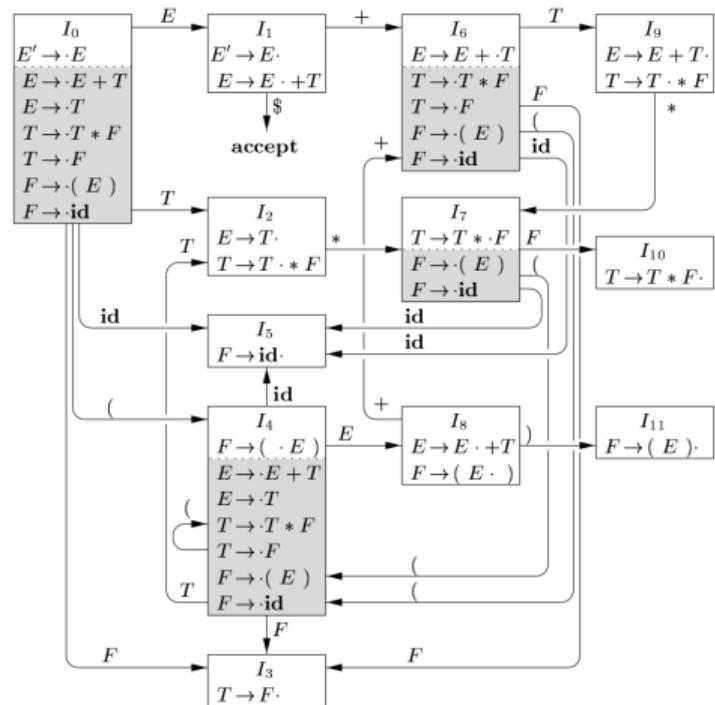
---

<sup>۱</sup> kernel items

<sup>۲</sup> nonkernel items

# Closure تابع

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



- یک تابع مهم دیگر تابع  $(X, I, \text{Goto})$  است. ورودی  $I$  مجموعه‌ای از آیتم‌هاست و  $X$  یک نماد از گرامر است.
- اگر در آیتم  $I$  داشته باشیم  $[A \rightarrow \alpha \cdot X\beta]$  آنگاه تابع  $\text{Goto}(I, X)$  برابر است با Closure برروی مجموعه همه آیتم‌های  $[A \rightarrow \alpha X \cdot \beta]$ .
- به طور شهودی، تابع Goto برای تعریف گذارها در ماشین LR( $^0$ ) برای یک گرامر استفاده می‌شود. حالت‌های ماشین مجموعه‌ای از آیتم‌هاست و  $(X, I, \text{Goto})$  گذار از حالت  $I$  با ورودی  $X$  است.

- اگر  $I$  مجموعه‌ای از دو آیتم  $\{[E' \rightarrow E\cdot], [E \rightarrow E \cdot + T]\}$  شامل آیتم‌های زیر است.

$$E \rightarrow E + \cdot T$$

$$T \rightarrow \cdot T * F$$

$$T \rightarrow \cdot F$$

$$F \rightarrow \cdot (E)$$

$$F \rightarrow \cdot \mathbf{id}$$

- برای محاسبه  $Goto(I, +)$  همه آیتم‌هایی را که در آنها  $+$  پس از نقطه قرار می‌گیرند در نظر می‌گیریم. آیتم  $E \rightarrow E \cdot + T$  چنین آیتمی است. نقطه را به بعد از  $+$  منتقل می‌کنیم و Closure آن را محاسبه می‌کنیم.

- الگوریتم زیر یک گروه استاندارد<sup>۱</sup> از مجموعه‌های آیتم‌های  $(^{\circ}LR)$  را برای گرامر افزوده شده  $G'$  می‌سازد.

```

void items( $G'$ ) {
     $C = \{\text{CLOSURE}(\{[S' \rightarrow \cdot S]\})\}$ ;
    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;
}

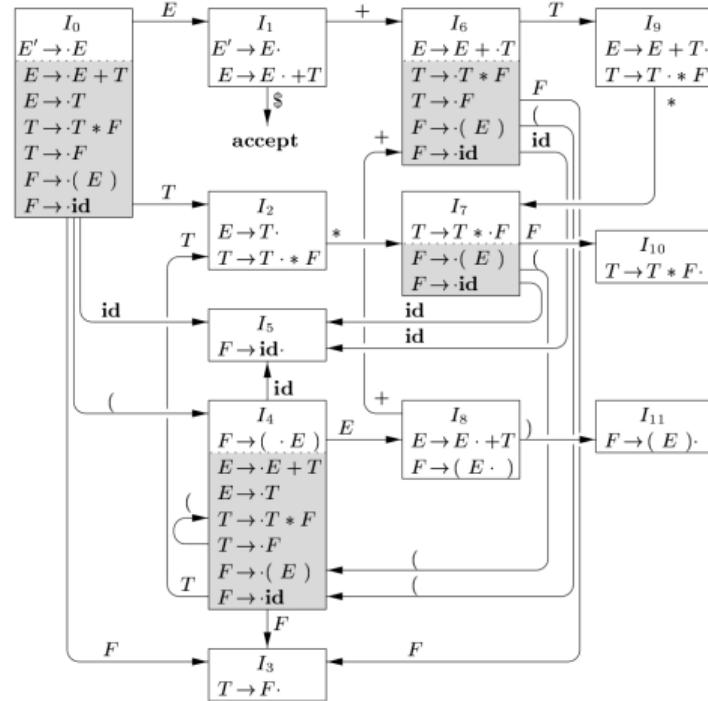
```

---

<sup>۱</sup> canonical collection

# Goto تابع

- گروه استاندارد از مجموعه‌های آیتم‌های LR( $\circ$ ) برای گرامر ذکر شده در شکل زیر نشان داده شده است. تابع Closure آیتم‌های خاکستری و تابع Goto گذارهای بین حالتها را محاسبه می‌کند.



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

تحلیل نحوی

# استفاده از ماشین $(^{\circ})LR$

- تجزیه LR ساده یا SLR<sup>1</sup> از ماشین  $(^{\circ})LR$  استفاده می‌کند.
- حالت‌های ماشین  $(^{\circ})LR$  مجموعه‌هایی از آیتم‌های گروه‌های  $LR(0)$  استاندارد<sup>2</sup> است و گذارها با تابع Goto محاسبه شده‌اند.

---

<sup>1</sup> simple LR

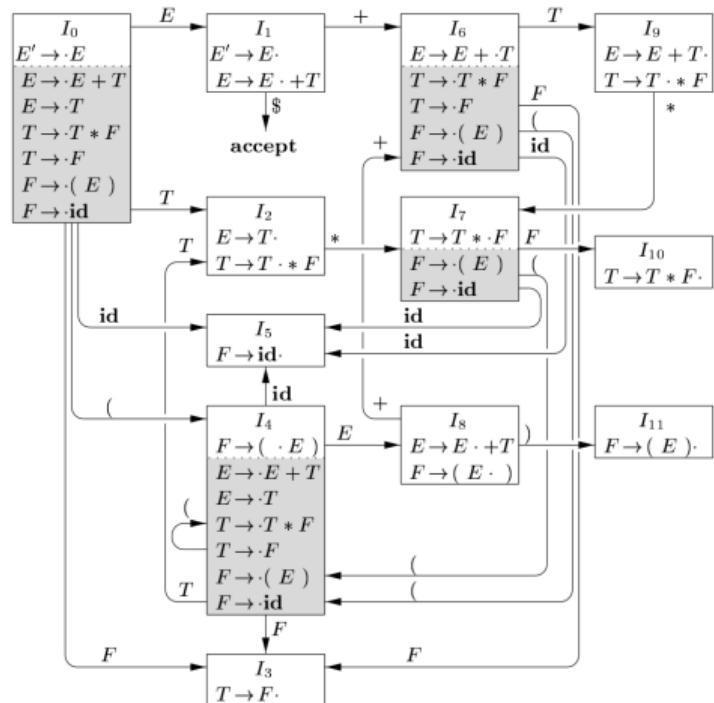
<sup>2</sup> sets of items from canonical  $LR(0)$  collection

## استفاده از ماشین ( $\circ$ )LR

- حالت آغازین ماشین ( $\circ$ )LR درواقع  $Closure([S' \rightarrow S] \cdot S)$  است، جایی که  $S'$  نماد آغازین گرامر افزوده شده است. همه حالت‌ها حالت پذیرش هستند.
- وقتی می‌گوییم حالت  $z$  منظور مجموعه آیتم‌های  $I_z$  است.
- حال باید ببینیم ماشین ( $\circ$ )LR چگونه کمک می‌کند برای انتقال کاهش تصمیم بگیریم.
- فرض کنید رشته  $a$  تشکیل شده از نمادهای گرامر، ماشین ( $\circ$ )LR را از حالت 0 به حالت  $z$  می‌برد. حال بروی نماد ورودی  $a$  انتقال انجام می‌دهیم اگر حالت  $z$  یک گذار با  $a$  دارد. در غیراینصورت یک کاهش انجام می‌دهیم که در اینصورت آیتم‌های حالت  $z$  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 $LR(\circ)$ مasha'in $(\circ)$

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



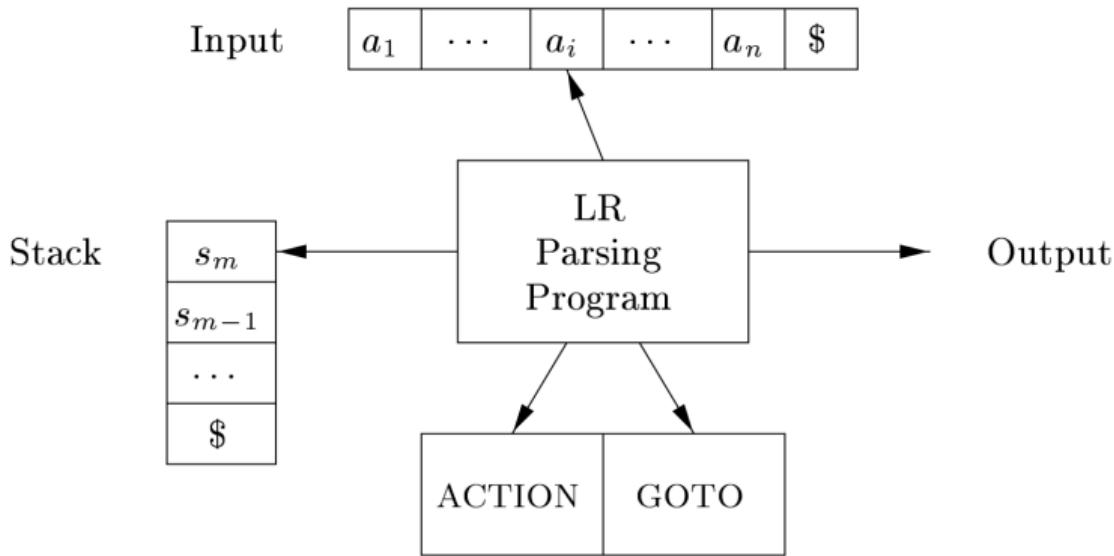
# استفاده از ماشین $LR(\circ)$

شكل زیر روند تجزیه  $id * id$  را نشان می‌دهد.

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	<b>id * id \$</b>	shift to 5
(2)	0 5	\$ <b>id</b>	* <b>id \$</b>	reduce by $F \rightarrow id$
(3)	0 3	\$ <b>F</b>	* <b>id \$</b>	reduce by $T \rightarrow F$
(4)	0 2	\$ <b>T</b>	* <b>id \$</b>	shift to 7
(5)	0 2 7	\$ <b>T *</b>	<b>id \$</b>	shift to 5
(6)	0 2 7 5	\$ <b>T * id</b>	\$	reduce by $F \rightarrow id$
(7)	0 2 7 10	\$ <b>T * F</b>	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ <b>T</b>	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ <b>E</b>	\$	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.
- برنامه تجزیه‌کننده برای همه تجزیه‌کننده‌های پایین به بالا 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 از یک حالت به حالت دیگر گذار می‌کند.

# الگوریتم تجزیه LR

- پشته شامل حالت‌های  $s_m \dots s_1 s_0$  می‌شود به طوری که  $s_m$  روی پشته است. پشته حالت‌های ماشین  $LR(\circ)$  را نگه می‌دارد.
- حالت‌ها متناظر هستند با مجموعه آیتم‌ها و یک گذار از حالت  $n$  به حالت  $j$  وجود دارد اگر  $. Goto(I_i, X) = I_j$
- همه گذارها به حالت  $j$  برای نماد  $X$  هستند. بنابراین هر حالت 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.
  - انتقال به حالت  $j$  : حالت  $j$  که نماینده ورودی  $a$  است، وارد پشته می‌شود.
  - کاهش  $\beta \rightarrow A$  : جمله  $\beta$  که برروی پشته قرار دارد با  $A$  کاهش پیدا می‌کند.
  - پذیرش : رشته پذیرفته می‌شود و تجزیه به اتمام می‌رسد.
  - خطأ : تجزیه‌کننده یک خطأ در ورودی می‌یابد و عملیات بازیابی خطأ 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  $j$  می‌برد.

## پیکربندی تجزیه کننده LR

- برای توصیف رفتار تجزیه کننده LR از یک نشانه گذاری برای نشان دادن وضعیت تجزیه کننده استفاده می کنیم، یعنی وضعیت پشته و رشتہ باقیمانده برای تجزیه.
- یک پیکربندی<sup>۱</sup> از تجزیه کننده LR یک دوتایی به صورت  $(s_0s_1 \dots s_m, a_i a_{i+1} \dots a_n \$)$  است، به طوری که جزء اول محتوای پشته (بالای پشته در سمت راست) و جزء دوم باقیمانده رشتہ ورودی است.
- این پیکربندی نشان دهنده صورت جمله ای  $X_m a_i a_{i+1} \dots a_n X_1 X_2 \dots X_l$  است.
- درواقع  $X_i$  نماد گرامری است که با حالت  $s_i$  نمایش داده می شود.
- توجه کنید که حالت آغازین  $s_0$  در تجزیه کننده نماینده هیچ نمادی از گرامر نیست و تنها زیر پشته را نشان می دهد.

---

<sup>1</sup> configuration

- حرکت بعدی تجزیه کننده از یک پیکربندی با خواندن نماد ورودی  $a_i$  و حالت  $s_m$  بروی پشته توسط Action $[s_m, a_i]$  از جدول تجزیه تعیین می شود.
- ۱. اگر  $s = \text{shift } s, a_i = \text{shift } s$  باشد، آنگاه تجزیه کننده یک عملیات انتقال انجام می دهد و حالت بعدی  $s$  را به پشته منتقل می کند و در پیکربندی  $(s_0s_1 \dots s_m, a_{i+1} \dots a_n \$)$  قرار می گیرد. نیازی نیست نماد  $a_i$  بروی پشته باشد زیرا اگر نیاز به آن بود می توان توسط حالت  $s$  آن را بازیابی کرد (البته در عمل هیچ گاه نیازی به آن نیست). نماد بعدی  $a_{i+1}$  خواهد بود.

## رفتار تجزیه کننده LR

۲. اگر  $\text{Action}[s_m, a_i] = \text{reduce } A \rightarrow \beta$  باشد، آنگاه تجزیه کننده 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  $\beta$  است و  $s = \text{Goto}[s_{m-r}, A]$ . در اینجا تجزیه کننده ابتدا تعداد  $r$  حالت را از پشته برمی‌دارد و حالت  $s_{m-r}$  برروی پشته قرار می‌گیرد. سپس حالت  $s$  یعنی  $\text{Goto}[s_{m-r}, A]$  برروی پشته قرار می‌گیرد. دنباله نمادهای  $X_{m-r+1} \cdots X_m$  متناظر است با حالت‌های برداشته شده از روی پشته که بر  $\beta$  یعنی بدنه قانون کاهش منطبق می‌شود.
۳. اگر  $\text{Action}[s_m, a_i] = \text{accept}$  آنگاه تجزیه به پایان می‌رسد.
۴. اگر  $\text{Action}[s_m, a_i] = \text{error}$  آنگاه تجزیه کننده یک تابع بازیابی کننده خطا فراخوانی می‌کند.

## الگوریتم تجزیه LR

- الگوریتم تجزیه‌کننده LR به صورت زیر عمل می‌کند. همه 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 خطأ صادر می‌شود.

# الگوريتم تجزيه LR

- در ابتدا تجزيه کننده  $s_0$  را ببروی پشته قرار می دهد و  $w\$$  ببروی بافر ورودی قرار می گيرد. سپس الگوريتم زير اجرا می شود.

```
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  $A \rightarrow \beta$  ) {
        pop  $|\beta|$  symbols off the stack;
        let state  $t$  now be on top of the stack;
        push GOTO[ $t, A$ ] onto the stack;
        output the production  $A \rightarrow \beta$ ;
    } else if ( ACTION[ $s, a$ ] = accept ) break;
    else call error-recovery routine;
}
```

# الگوریتم تجزیه LR

- توابع Action و Goto از تجزیه کننده LR برای گرامر ذکر شده در زیر نشان محسوبه شده‌اند.

STATE	ACTION					GOTO		
	<b>id</b>	+	*	( )	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4		1	2	3
1		s6			acc			
2		r2	s7		r2	r2		
3		r4	r4		r4	r4		
(1) $E \rightarrow E + T$	4	s5		s4		8	2	3
(2) $E \rightarrow T$	5		r6	r6	r6	r6		
(3) $T \rightarrow T * F$	6	s5		s4			9	3
	7	s5		s4				10
(4) $T \rightarrow F$	8		s6		s11			
(5) $F \rightarrow (E)$	9		r1	s7	r1	r1		
(6) $F \rightarrow \text{id}$	10		r3	r3	r3	r3		
	11		r5	r5	r5	r5		

# الگوریتم تجزیه LR

- در این جدول  $i$  به معنی انتقال به حالت  $n$  است،  $j$  به معنی کاهش با قانون شماره  $j$  است،  $acc$  به معنی پذیرش و خانه‌های خالی به معنی خطأ است.

# الگوریتم تجزیه LR

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

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		<b><math>id * id + id \\$</math></b>	shift
(2)	0 5	<b><math>id</math></b>	* $id + id \$$	reduce by $F \rightarrow id$
(3)	0 3	<b><math>F</math></b>	* $id + id \$$	reduce by $T \rightarrow F$
(4)	0 2	<b><math>T</math></b>	* $id + id \$$	shift
(5)	0 2 7	$T *$	<b><math>id + id \\$</math></b>	shift
(6)	0 2 7 5	$T * id$	+ $id \$$	reduce by $F \rightarrow id$
(7)	0 2 7 10	$T * F$	+ $id \$$	reduce by $T \rightarrow T * F$
(8)	0 2	$T$	+ $id \$$	reduce by $E \rightarrow T$
(9)	0 1	$E$	+ $id \$$	shift
(10)	0 1 6	$E +$	<b><math>id \\$</math></b>	shift
(11)	0 1 6 5	$E + id$	\$	reduce by $F \rightarrow id$
(12)	0 1 6 3	$E + F$	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E + T$	\$	reduce by $E \rightarrow E + T$
(14)	0 1	$E$	\$	accept

- برای مثال در خط (۱) تجزیه‌کننده در حالت ۰ است و اولین نماد  $s_5$  باید انجام شود، بدین معنی که یک انتقال با وارد کردن حالت ۵ به پشته انجام می‌شود. سپس \* نماد بعدی است و عملیات حالت ۵ بر روی ورودی \* یک کاوش با  $id \rightarrow F$  است. یک حالت از پشته برداشته می‌شود. چون Goto در حالت ۰ بر روی F حالت ۳ است، پس حالت ۳ بر روی 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  $LR(\cdot)$  و ماشین  $LR(\cdot)$  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 الگوریتم زیر ساخته می‌شود. قبل از ساختن جدول نیاز داریم برای همه 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  $(^0)LR$  برای گرامر  $G$  به صورت  $\{I_0, I_1, \dots, I_n\}$  می‌سازیم.
  - ۲. حالت  $i$  را به ازای  $I_i$  می‌سازیم و عملیات تجزیه برای حالت  $i$  را به صورت زیر تعیین می‌کنیم :
    - اگر  $[A \rightarrow \alpha \cdot a\beta]$  در  $I_i$  باشد و  $a$  آنگاه  $j$  باشد،  $Goto(I_i, a) = I_j$  . در اینجا  $a$  باید یک ترمینال باشد.
    - اگر  $[A \rightarrow \alpha \cdot]$  در  $I_i$  باشد، آنگاه  $a$  در  $Follow(A)$  به ازای هر  $a$  در  $I_i$  باشد،  $Goto(I_i, a) = reduce A \rightarrow \alpha \cdot a\beta$  باشد.
    - اگر  $[S' \rightarrow S \cdot]$  در  $I_i$  باشد آنگاه  $Goto(I_i, \$) = accept$  باشد.

- اگر در حین اجرای این الگوریتم تعارضی در عملیات به وجود آمد، می‌گوییم گرامر (1) SLR(1) نیست و الگوریتم نمی‌تواند تجزیه‌کننده تولید کند.
- ۳. گذار Goto برای هر حالت  $i$  برای همه متغیرهای  $A$  با استفاده از این قانون محاسبه می‌شود : اگر  $Goto[i, A] = j$  آنگاه  $Goto(I_i, A) = I_j$
- 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  $[S] \rightarrow [S']$  است.

# ساختن جدول تجزیه LR

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

## ساختن جدول تجزیه LR

– می‌خواهیم جدول تجزیه SLR برای گرامر زیر بسازیم.

$$(1) \quad E \rightarrow E + T$$

$$(2) \quad E \rightarrow T$$

$$(3) \quad T \rightarrow T * F$$

$$(4) \quad T \rightarrow F$$

$$(5) \quad F \rightarrow (E)$$

$$(6) \quad F \rightarrow \text{id}$$

– مجموعه آیتم‌های  $I_0$  به صورت زیر است.

$$E' \rightarrow \cdot E$$

$$E \rightarrow \cdot E + T$$

$$E \rightarrow \cdot T$$

$$T \rightarrow \cdot T * F$$

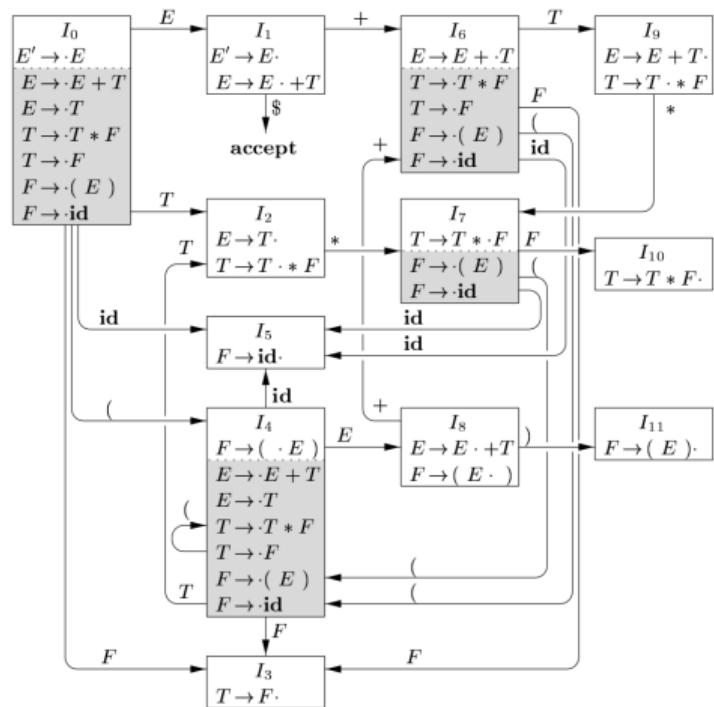
$$T \rightarrow \cdot F$$

$$F \rightarrow \cdot (E)$$

$$F \rightarrow \cdot \text{id}$$

# ساختن جدول تجزیه LR

- مجموعه آیتمها را به صورت زیر قبل محاسبه کردیم.



## ساختن جدول تجزیه LR

- با استفاده از آیتم  $(E \cdot F) \rightarrow E \cdot \text{shift } 4$  می‌توان مقدار  $\text{Action}[0, ()] = \text{shift } 4$  را محاسبه کرد و با استفاده از آیتم  $(E \cdot F) \rightarrow E \cdot \text{id}$  مقدار  $\text{Action}[0, \text{id}] = \text{shift } 5$  مقداری به دست نمی‌دهند.
  - حال آیتم‌های  $I_1$  را در نظر می‌گیریم. برای  $E' \rightarrow E \cdot$  به دست می‌آوریم  $\text{Action}[1, \$] = \text{accept}$  و برای  $E \rightarrow E \cdot + T$  به دست می‌آوریم  $\text{Action}[1, +] = \text{shift } 6$ .
  - برای آیتم‌های  $I_2$  نیز عملیات را محاسبه می‌کنیم. برای آیتم  $T \rightarrow E \cdot$  از آنجایی که  $\text{Follow}(E) = \{\$, +, ()\}$  داریم:
- $$\text{Action}[2, \$] = \text{Action}[2, +] = \text{Action}[2, ()] = \text{reduce } E \rightarrow T$$
- $$\cdot \text{Action}[2, *] = \text{shift } 7 \quad \text{در } I_2 \text{ داریم}$$
- همچنین برای آیتم  $T \rightarrow T \cdot * F$  داریم

# ساختن جدول تجزیه LR

جدول تجزیه برای این گرامر به صورت زیر محاسبه می‌شود.

STATE	ACTION					GOTO		
	<b>id</b>	+	*	( )	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4		1	2	3
1		s6			acc			
2		r2	s7		r2	r2		
3		r4	r4		r4	r4		
(1) $E \rightarrow E + T$	4	s5		s4		8	2	3
(2) $E \rightarrow T$	5		r6	r6	r6	r6		
(3) $T \rightarrow T * F$	6	s5		s4			9	3
	7	s5		s4				10
(4) $T \rightarrow F$	8		s6		s11			
(5) $F \rightarrow (E)$	9		r1	s7	r1	r1		
(6) $F \rightarrow \text{id}$	10		r3	r3	r3	r3		
	11		r5	r5	r5	r5		

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

$$\begin{array}{lcl} S & \rightarrow & L = R \quad | \quad R \\ L & \rightarrow & *R \quad | \quad \text{id} \\ R & \rightarrow & L \end{array}$$

# ساختن جدول تجزیه LR

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

$$\begin{aligned} I_0: \quad & S' \rightarrow \cdot S \\ & S \rightarrow \cdot L = R \\ & S \rightarrow \cdot R \\ & L \rightarrow \cdot * R \\ & L \rightarrow \cdot \text{id} \\ & R \rightarrow \cdot L \end{aligned}$$

$$\begin{aligned} I_5: \quad & L \rightarrow \text{id} \cdot \\ I_6: \quad & S \rightarrow L = \cdot R \\ & R \rightarrow \cdot L \\ & L \rightarrow \cdot * R \\ & L \rightarrow \cdot \text{id} \end{aligned}$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$I_7: \quad L \rightarrow *R \cdot$$

$$\begin{aligned} I_2: \quad & S \rightarrow L \cdot = R \\ & R \rightarrow L \cdot \end{aligned}$$

$$I_8: \quad R \rightarrow L \cdot$$

$$I_3: \quad S \rightarrow R \cdot$$

$$I_9: \quad S \rightarrow L = R \cdot$$

$$\begin{aligned} I_4: \quad & L \rightarrow * \cdot R \\ & R \rightarrow \cdot L \\ & L \rightarrow \cdot * R \\ & L \rightarrow \cdot \text{id} \end{aligned}$$

# ساختن جدول تجزیه LR

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

- حال ببینیم چرا ماشین  $(^0) LR$  می‌تواند برای تصمیم‌گیری درمورد انتقال و کاهش استفاده شود.
- در ماشین  $(^0) LR$  برای یک گرامر، پشته، پیشوندی از صورت جمله‌ای راست را نگهداری می‌کند.
- اگر پشته  $\alpha$  را نگهداری کند و ورودی باقیمانده  $x$  باشد آنگاه دنباله‌ای از کاهش‌ها  $\alpha x$  را به  $S$  کاهش می‌دهد. درواقع در فرایند اشتقاء داریم  $\alpha x \xrightarrow[^*]{rrm} S$ .
- اما همه پیشوندهای صورت‌های جمله‌ای راست نمی‌توانند بروی پشته ظاهر شوند. برای مثال فرض کنید داشته باشیم  $id * id \xrightarrow[^*]{rrm} E * id \xrightarrow[^*]{rrm} F * id$  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.
- همچنین قبل اثبات کردیم که از آنجایی که فرایند کاهش معکوس فرایند اشتقاء راست است، هندل نمی‌تواند در وسط پشته باشد و حتماً باید بر روی پشته قرار بگیرد.

## پیشوندهای پایدار

- پیشوندهایی از صورت‌های جمله‌ای راست که می‌توانند ببروی پشته دریک تجزیه‌کننده انتقال کاهش ظاهر شوند، پیشوندهای پایدار<sup>۱</sup> نامیده می‌شوند. یک پیشوند پایدار پیشوندی از یک صورت جمله‌ای راست است که یک هندل را شامل نمی‌شود.
- تجزیه SLR بر این اصل عمل می‌کند که ماشین  $(\circ)LR$  پیشوندهای پایدار را تشخیص می‌دهد. می‌گوییم آیتم  $A \rightarrow \beta_1 \cdot \beta_2$  برای پیشوند پایدار  $\alpha\beta_1$  معتبر است اگر اشتاقاقی به صورت  $S' \xrightarrow{^*_{rm}} \alpha Aw \xrightarrow{^*_{rm}} \alpha\beta_1\beta_2w$  وجود داشته باشد. یک آیتم می‌تواند برای بسیاری از پیشوندهای پایدار معتبر باشد.
- این که  $A \rightarrow \beta_1 \cdot \beta_2$  برای  $\alpha\beta_1$  معتبر است به ما می‌گوید وقتی  $\alpha\beta_1$  را ببروی پشته مشاهده کردیم انتقال انجام دهیم یا کاهش.
- اگر  $\epsilon \neq \beta_2$  آنگاه می‌توان نتیجه گرفت که هندل به پشته انتقال داده نشده است بنابراین باید انتقال انجام شود. اگر  $\epsilon = \beta_2$  آنگاه  $A \rightarrow \beta_1$  باید هندل باشد و باید با استفاده از این قانون کاهش انجام شود.

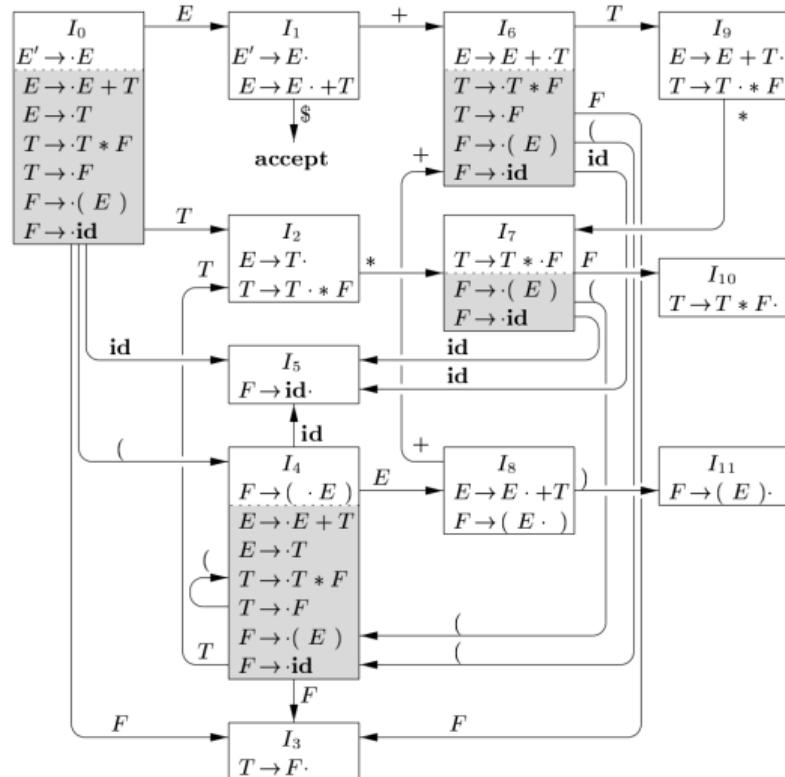
<sup>1</sup> viable prefix

## پیشوندهای پایدار

- مجموعه آیتم‌های معتبر برای پیشوند پایدار ۷، مجموعه آیتم‌هایی است که از حالت اولیه با مسیری با برچسب  $\gamma$  در ماشین  $(^{\circ}LR)$  قابل دسترسی هستند.
- مجموعه همه آیتم‌های معتبر همه اطلاعات مهم که می‌توانند در پشته قرار بگیرند را در برمی‌گیرد. این قضیه در نظریه تجزیه‌کننده LR اثبات می‌شود که به اثبات آن نمی‌پردازیم.

# پیشوندهای پایدار

- گرامر زیر و مجموعه آیتم‌های این گرامر به همراه توابع گذار را که به صورت زیر محاسبه کردیم، در نظر بگیرید.



## پیشوندهای پایدار

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

$$T \rightarrow T \cdot F$$

$$F \rightarrow \cdot(E)$$

$$F \rightarrow \cdot.id$$

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

$$\begin{array}{l} E' \Rightarrow E \\ \quad \stackrel{rm}{\Rightarrow} E + T \\ \quad \stackrel{rm}{\Rightarrow} E + T * F \\ \quad \stackrel{rm}{\Rightarrow} E + T * (E) \end{array}$$

$$\begin{array}{l} E' \Rightarrow E \\ \quad \stackrel{rm}{\Rightarrow} E + T \\ \quad \stackrel{rm}{\Rightarrow} E + T * F \\ \quad \stackrel{rm}{\Rightarrow} E + T * (E) \end{array}$$

$$\begin{array}{l} E' \Rightarrow E \\ \quad \stackrel{rm}{\Rightarrow} E + T \\ \quad \stackrel{rm}{\Rightarrow} E + T * F \\ \quad \stackrel{rm}{\Rightarrow} E + T * id \end{array}$$

- اشتقاق اول معتبر بودن  $F \cdot T \rightarrow T^* E$  را نشان می‌دهد. اشتقاق دوم معتبر بودن  $(E) \cdot F \rightarrow F$  و اشتقاق سوم معتبر بودن  $.id \cdot F \rightarrow F$  را نشان می‌دهد.

- می‌توان نشان داد که هیچ آیتم معتبر دیگری برای  $E + T^* E$  وجود ندارد.

## تجزیه‌کننده‌های LR قدرتمندتر

- در این قسمت تجزیه‌کننده‌های LR را تعمیم می‌دهیم و دو تجزیه‌کننده قدرتمندتر را شرح می‌دهیم.
- ۱. تجزیه‌کننده LR استاندارد<sup>۱</sup> یا CLR : این تجزیه‌کننده از مجموعه‌ای بزرگ از آیتم‌ها به نام آیتم‌های (1) استفاده می‌کند.
- ۲. تجزیه‌کننده LR با بررسی نماد بعدی<sup>۲</sup> یا LALR : این تجزیه‌کننده تعداد بسیار کمتری حالت نسبت به تجزیه‌کننده‌ها بر پایه (1) LR 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.

---

<sup>۱</sup> Canonical LR (CLR)

<sup>۲</sup> Look Ahead LR (LALR)

## آیتم‌های LR(1) استاندارد

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

# آیتم‌های LR(1) استاندارد

- مثال : گرامر زیر را در نظر بگیرید.

$$\begin{array}{lcl} S & \rightarrow & L = R \quad | \quad R \\ L & \rightarrow & *R \quad | \quad \text{id} \\ R & \rightarrow & L \end{array}$$

# آیتم‌های LR(1) استاندارد

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

$$\begin{aligned} I_0: \quad S' &\rightarrow \cdot S \\ &S \rightarrow \cdot L = R \\ &S \rightarrow \cdot R \\ &L \rightarrow \cdot * R \\ &L \rightarrow \cdot \mathbf{id} \\ &R \rightarrow \cdot L \end{aligned}$$

$$\begin{aligned} I_5: \quad L &\rightarrow \mathbf{id} \cdot \\ I_6: \quad S &\rightarrow L = \cdot R \\ &R \rightarrow \cdot L \\ &L \rightarrow \cdot * R \\ &L \rightarrow \cdot \mathbf{id} \end{aligned}$$

$$I_1: \quad S' \rightarrow S \cdot$$

$$I_7: \quad L \rightarrow * R \cdot$$

$$\begin{aligned} I_2: \quad S &\rightarrow L \cdot = R \\ &R \rightarrow L \cdot \end{aligned}$$

$$\begin{aligned} I_8: \quad R &\rightarrow L \cdot \\ I_9: \quad S &\rightarrow L = R \cdot \end{aligned}$$

$$I_3: \quad S \rightarrow R \cdot$$

$$\begin{aligned} I_4: \quad L &\rightarrow * \cdot R \\ &R \rightarrow \cdot L \\ &L \rightarrow \cdot * R \\ &L \rightarrow \cdot \mathbf{id} \end{aligned}$$

## آیتم‌های LR(1) استاندارد

- گرچه نماد = در  $\text{Follow}(R)$  است، و در فرایند اشتقاق می‌توانیم داشته باشیم  
 $S \Rightarrow L = R \Rightarrow *R = R$  وجود ندارد. پس اگر در پشته  $L$  داشته باشیم و علامت = در ورودی خوانده شود، باید کاهش توسط  $L \rightarrow R$ ، انجام شود.
- در حالت ۲ آیتم  $L \rightarrow R$  را داریم. فرض کنید در این حالت، در ورودی علامت = را می‌خوانیم که در  $\text{Follow}(R)$  است.
- تجزیه‌کننده SLR در حالت ۲ با خواندن نماد = می‌تواند کاهش با استفاده از  $L \rightarrow R$  را انجام می‌دهد.
- هیچ صورت جمله‌ای در این گرامر که با ... =  $R$  آغاز شود وجود ندارد. بنابراین در چنین شرایطی در حالت ۲ باید  $L$  را با  $R$  کاهش داد.

## 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 حالت‌های تجزیه‌کننده LR مشخص کنیم کدام نمادها می‌توانند هندل  $\alpha$  را دنبال کنند وقتی یک کاهش به صورت  $\alpha \rightarrow A$  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. مؤلفه دوم آیتم، نماد بعدی<sup>۱</sup> آیتم نیز نامیده می‌شود.
- نماد بعدی (مؤلفه دوم) یک آیتم، هیچ تأثیری در آیتمی که به صورت  $[A \rightarrow \alpha \cdot \beta, a]$  است ندارد اگر  $\beta$  رشته تهی نباشد. اما در آیتم  $[A \rightarrow \alpha \cdot, a]$  کاهش با استفاده از  $A \rightarrow \alpha$  تنها صورتی انجام می‌شود که نماد بعدی  $a$  باشد.
- می‌گوییم آیتم LR(1) به صورت  $[A \rightarrow \alpha \cdot \beta, a]$  برای پیشوند  $\delta\alpha$  معتبر است اگر اشتقةایی به صورت  $S \xrightarrow[\text{rm}]{*} \delta Aw \xrightarrow[\text{rm}]{*} \delta\alpha\beta w$  وجود داشته باشد و  $a$  اولین نماد  $w$  باشد و یا  $w$  تهی باشد و  $a$  نماد  $\$$  باشد.

---

<sup>1</sup> lookahead

## آیتم‌های LR(1) استاندارد

- مثال : گرامر زیر را در نظر بگیرید.

$$\begin{aligned} S &\rightarrow B \ B \\ B &\rightarrow a \ B \quad | \quad b \end{aligned}$$

- اشتقاق راست  $S \xrightarrow[\text{rm}]{*} aaBab \xrightarrow[\text{rm}]{*} aaaBab$  برای پیشوند  $[B \rightarrow a \cdot B, a]$  وجود دارد. در اینجا آیتم  $\gamma = aaa$  معتبر است.

- همچنین اشتقاق راست  $S \xrightarrow[\text{rm}]{*} BaB \xrightarrow[\text{rm}]{*} BaaB$  برای پیشوند  $[B \rightarrow a \cdot B, \$]$  وجود دارد. در این فرایند اشتقاق، آیتم  $\gamma = Baa$  معتبر است.

## ساختن مجموعه‌های آیتم‌های LR(1)

- روش ساخت مجموعه آیتم‌های LR(1) شبیه ساخت مجموعه آیتم‌های  $(LR)^0$  است. تنها تفاوت در توابع Goto و Closure 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  $[A \rightarrow \alpha \cdot B \beta, a]$  in  $I$  )
            for ( each production  $B \rightarrow \gamma$  in  $G'$  )
                for ( each terminal  $b$  in FIRST( $\beta a$ ) )
                    add  $[B \rightarrow \cdot \gamma, b]$  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  $[A \rightarrow \alpha \cdot X \beta, a]$  in  $I$  )
        add item  $[A \rightarrow \alpha X \cdot \beta, a]$  to set  $J$ ;
    return CLOSURE( $J$ );
}
```

# ساختن مجموعه‌های آیتم‌های LR(1)

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

```
void items( $G'$ ) {
    initialize  $C$  to {CLOSURE( $\{[S' \rightarrow \cdot S, \$]\}$ )};
    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 محاسبه  $\text{Closure}([A \rightarrow \alpha \cdot B\beta, a])$  به ازای همه ترمینال‌های  $b$  در  $\text{First}(\beta a)$  باید آیتم  $[B \rightarrow \gamma \cdot b]$  را اضافه کنیم.
- این آیتم بدین معناست که در فرایند کاهش به متغیر  $A$  اگر در پشته  $\alpha$  قرار داشته باشد و به متغیر  $B$  برسیم، ابتدا باید ورودی را توسط یکی از قوانین متعلق به  $B$  کاهش دهیم. اما پس از کاهش ورودی به  $B$  آنچه در ورودی قرار می‌گیرد، ترمینالی است که بعد از متغیر  $B$  در  $\text{First}(\beta)$  قرار دارد و اگر  $\beta$  تبدیل به تهی شود، آنچه بعد از  $B$  در ورودی قرار می‌گیرد ترمینال  $a$  است.
- پس می‌گوییم هر یک از ترمینال‌های  $b$  در  $\text{First}(\beta a)$  پس از کاهش ورودی به  $B$  می‌تواند در ورودی خوانده می‌شود.

## ساختن مجموعه‌های آیتم‌های LR(1)

- مثال : گرامر زیر را در نظر بگیرید.

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

- ابتدا  $[A \rightarrow \alpha \cdot B\beta, a]$  Closure( $[S' \rightarrow \cdot S, \$]$ ) را محاسبه می‌کنیم. آیتم  $[S' \rightarrow \cdot S, \$]$  را برابر با  $\text{Closure}([S' \rightarrow \cdot S, \$])$  دانیم. پس  $S'$  را برابر با  $\text{First}(\beta a)$  دانیم. این گرامر را به ازای هر  $b$  در  $\text{First}(\beta a)$  اضافه می‌کنیم. در این گرامر  $B \rightarrow \gamma \cdot b$  قانون  $\gamma \rightarrow \gamma \cdot b$  است، و از آنجایی که  $\beta$  تهی و  $a$  نماد  $\$$  است، پس  $b$  تنها می‌تواند  $\$$  باشد. پس آیتم  $[S \rightarrow \cdot CC, \$]$  را می‌افزاییم.

## ساختن مجموعه‌های آیتم‌های LR(1)

- سپس همه آیتم‌های  $[C \rightarrow \cdot b, b]$  را برای  $b$  در  $\text{First}(C\$)$  اضافه می‌کنیم. از آنجایی که  $C$  رشته تهی تولید نمی‌کند،  $\text{First}(C\$) = \text{First}(C) = \{c, d\}$ . بنابراین، آیتم‌های  $[C \rightarrow \cdot cC, c]$  و  $[C \rightarrow \cdot cC, d]$  و  $[C \rightarrow \cdot d, d]$  را می‌افزاییم. هیچ‌کدام از این آیتم‌ها در سمت راست نقطه متغیر ندارد، بنابراین محاسبه اولین مجموعه LR(1) به اتمام می‌رسد.

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

- در اینجا برای سادگی به جای دو آیتم  $[C \rightarrow \cdot cC, c]$  و  $[C \rightarrow \cdot cC, d]$  می‌نویسیم.

## ساختن مجموعه‌های آیتم‌های LR(1)

- حال باید  $Goto(I_0, X)$  را برای مقادیر مختلف  $X$  محاسبه کنیم. اگر  $S = S'$  باشد آیتم  $[S' \rightarrow S\cdot, \$]$  به دست می‌آید. بنابراین داریم.

$$I_1 : S' \rightarrow S\cdot, \$$$

- برای  $C$  آیتم  $X = C$  به وجود می‌آید. همه قوانین متغیر  $C$  را با نماد  $\$$  به عنوان مؤلفه دوم می‌افزاییم و به دست می‌آوریم :

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

## ساختن مجموعه‌های آیتم‌های LR(1)

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

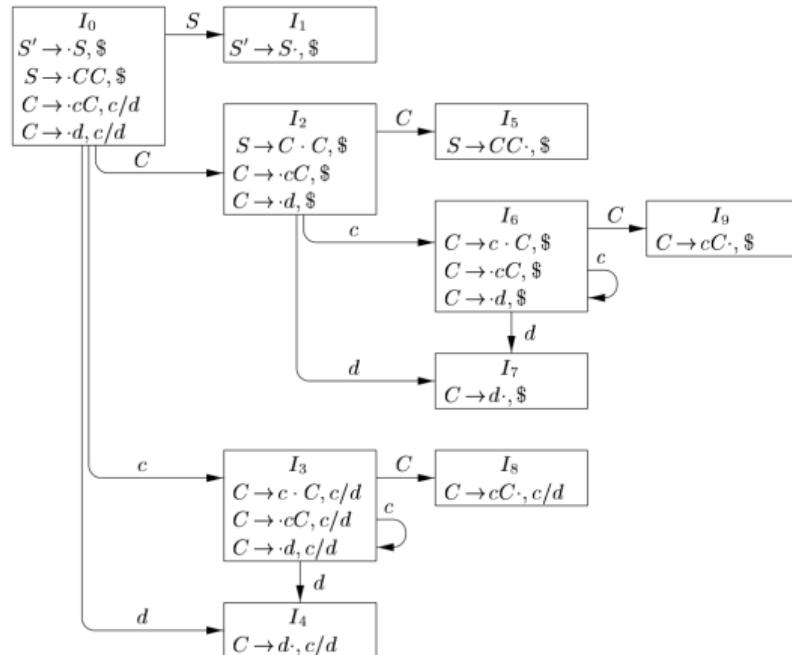
$$\begin{aligned} I_3 : \quad & C \rightarrow c \cdot C, \quad c/d \\ & C \rightarrow \cdot cC, \quad c/d \\ & C \rightarrow \cdot d, \quad c/d \end{aligned}$$

- در نهایت قرار می‌دهیم  $d = X$  و به دست می‌آوریم.

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

# ساختن مجموعه‌های آیتم‌های LR(1)

- به همین ترتیب با محاسبه Closure گراف زیر را به دست می‌آوریم.



## جداول تجزیه (۱) استاندارد LR

- 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(۱) را به صورت  $\{I_0, I_1, \dots, I_n\}$  می‌سازیم.
  ۲. حالت  $i$  از تجزیه‌کننده از  $I_i$  ساخته می‌شود. عملیات تجزیه برای  $i$  به صورت زیر تعیین می‌شود.
    - اگر  $Action[i, a] = shift$  باشد، آنگاه  $j = Goto(I_i, a)$  در  $[A \rightarrow \alpha \cdot a\beta, b]$  باشد و  $I_j$  باشد.
    - اگر  $Action[i, a] = reduce A \rightarrow \alpha \cdot$  باشد و  $A \neq S'$  در  $I_i$  باشد، آنگاه  $S' \rightarrow S \cdot$  باشد.
    - اگر  $Action[i, \$] = accept$  باشد، آنگاه  $S' \rightarrow S \cdot$  در  $I_i$  باشد.
    - اگر هرگونه تعارضی در عملیات بالا رخ دهد می‌گوییم گرامر LR(۱) نیست.

## جداول تجزیه LR(1) استاندارد

۳. توابع گذار Goto برای حالت  $i$  برای همه متغیرهای  $A$  به صورت زیر ساخته می‌شود : اگر
$$Goto[i, A] = j \quad Goto(I_i, A) = I_j$$
۴. همه خانه‌هایی که در گام‌های (۲) و (۳) تعریف نشده‌اند، خطأ محسوب می‌شوند.
۵. حالت اولیه تجزیه‌کننده، حالتی است که از مجموعه آیتم‌های حاوی  $[S' \rightarrow S, \$]$  تشکیل شده است.

# جداول تجزیه LR(1) استاندارد

- جدولی که با استفاده از الگوریتم قبل تولید می‌شود جدول تجزیه LR(1) استاندارد<sup>1</sup> نامیده می‌شود.
- یک تجزیه‌کننده 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.

---

<sup>1</sup> canonical LR(1)

# جداول تجزیه (۱) استاندارد LR(1)

جدول تجزیه (۱) استاندارد برای گرامر ذکر شده در زیر نشان داده شده است. قوانین ۱، ۲ و ۳ به ترتیب  $C \rightarrow d$ ,  $C \rightarrow cC$ ,  $S \rightarrow CC$  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 (۱) استاندارد LR

- هر گرامر (۱) SLR است اما برای یک گرامر (۱) LR تجزیه‌کننده LR استاندارد ممکن است تعداد حالت‌های بیشتری نسبت به تجزیه‌کننده SLR برای همان گرامر داشته باشد.

# ساختن جداول تجزیه LALR

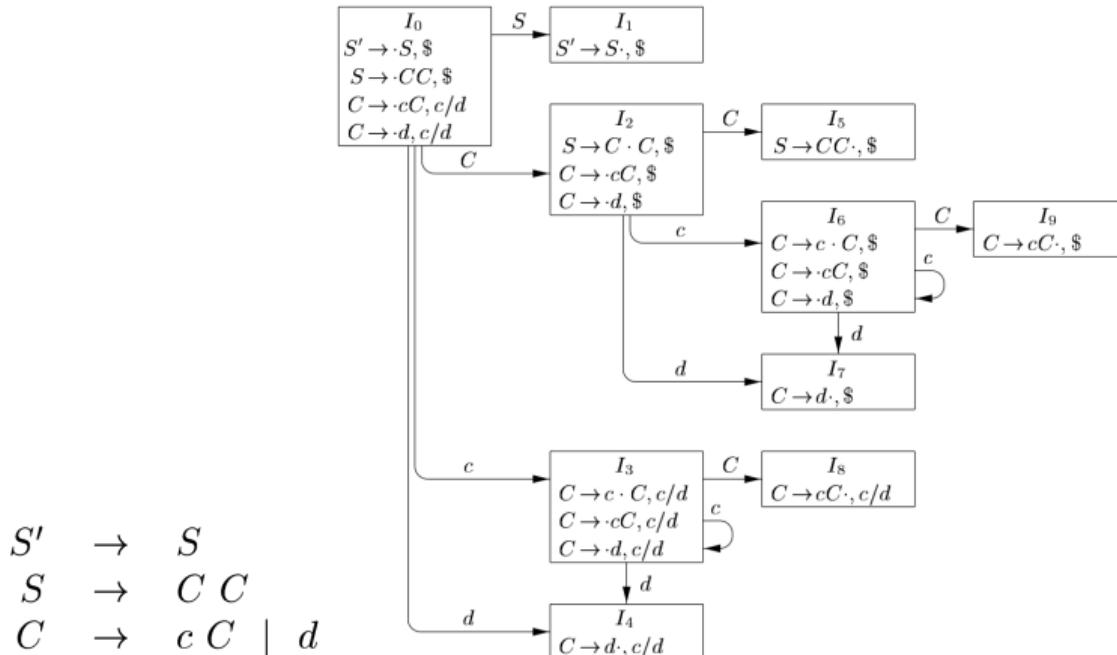
- حال به معرفی تجزیه‌کننده  $LALR^1$  می‌پردازیم. این تجزیه‌کننده در عمل بسیار مورد استفاده قرار می‌گیرد، زیرا جداول تجزیه آن از جدول تجزیه‌کننده LR استاندارد کوچک‌تر هستند و بیشتر زبان‌های برنامه‌نویسی را می‌توان با استفاده از گرامر LALR می‌توان توصیف کرد.
- جداول SLR نیز نسبت به CLR کوچک‌تر هستند اما برخی از ساختارهای زبان‌های برنامه‌نویسی را نمی‌توان با استفاده از گرامرهای SLR توصیف کرد.
- جداول تجزیه SLR و LALR تقریباً 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.

---

<sup>1</sup> Lookahead LR

# ساختن جداول تجزیه LALR

- گرامر زیر و مجموعه آیتم‌های LR(1) برای این گرامر را در نظر بگیرید.



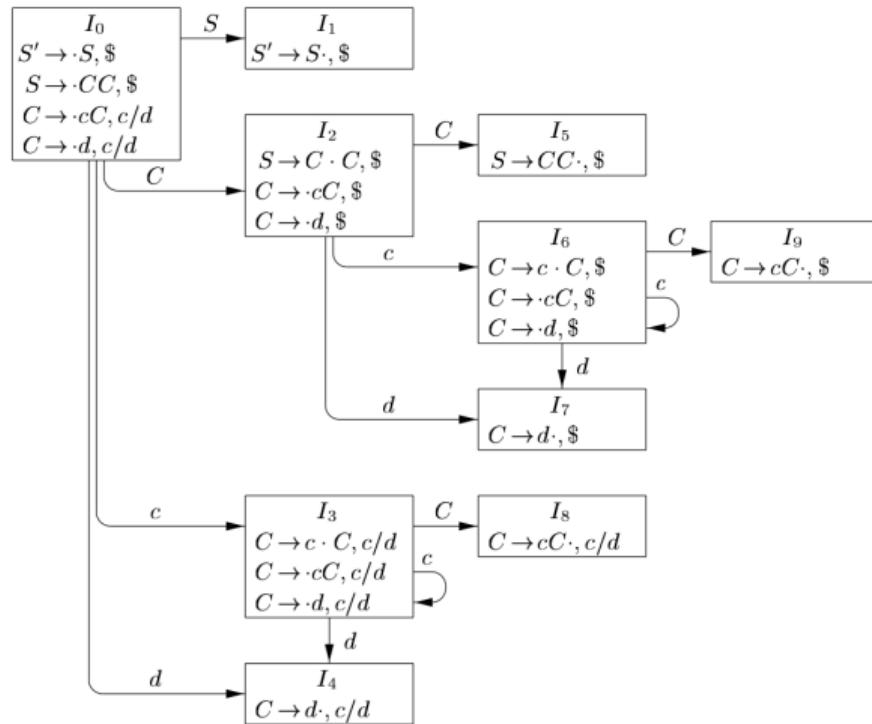
- دو حالت شبیه به هم  $I_4$  و  $I_7$  را در نظر بگیرید. هریک از این حالت‌ها فقط آیتم‌هایی با مؤلفه اول  $C \rightarrow d$  دارند. در  $I_4$  مؤلفه دوم  $c$  یا  $d$  است و در  $I_7$  مؤلفه دوم  $\$$  است.
- برای اینکه تفاوت  $I_4$  و  $I_7$  را بفهمیم، یک مثال را بررسی می‌کنیم. این گرامر زبان منظم  $c^*dc^*d$  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  $C \rightarrow d$  انجام می‌شود. با توجه به اینکه کاراکتر بعدی می‌تواند  $c$  یا  $d$  باشد، اگر  $\$$  به دنبال اولین  $d$  بیاید، یک ورودی به صورت  $cccd$  داریم که در زبان نیست و حالت ۴ به درستی با خواندن  $\$$  وجود خطا را نشان می‌دهد.

## ساختن جداول تجزیه LALR

- تجزیه کننده بعد از خواندن دومین  $d$  وارد حالت ۷ می‌شود. پس از آن باید در ورودی  $\$$  خوانده شود و گرنه ورودی بربطق الگوی  $c^*dc^*$  نیست. بنابراین در حالت ۷ با خواندن ورودی  $\$$  کاهش  $d \rightarrow C$  انجام می‌شود و با ورودی  $c$  یا  $d$  خطأ صادر می‌شود.
- حال فرض کنید  $I_4$  و  $I_7$  را با  $I_{47}$  جایگزین کنیم که اجتماع  $I_4$  و  $I_7$  است و از سه آیتم  $[C \rightarrow d \cdot, c/d/\$]$  تشکیل شده است.
- عملیات حالت ۴۷ اکنون این است که بر روی همه ورودی‌ها کاهش انجام می‌دهد، در حالی که قبل از ادغام دو حالت برخی از شرایط منجر به خطأ می‌شدند.
- ادغام این دو حالت تعداد حالت‌ها را کاهش می‌دهد. البته خطای ورودی بعد از ادغام دو حالت نیز تشخیص داده خواهد شد، اما شناسایی خطأ دیرتر انجام می‌شود.
- جهت کاهش تعداد حالات تجزیه کننده LR استاندارد، 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  $\{C \rightarrow d\}$  وجود دارد. همچنین در  $I_3$  و  $I_6$  آیتم‌هایی با هسته‌های  $\{C \rightarrow c \cdot C, C \rightarrow cC, C \rightarrow \cdot d\}$  وجود دارد. در  $I_8$  و  $I_9$  نیز هسته  $\{C \rightarrow c \cdot C, C \rightarrow \cdot d\}$  وجود دارد.
- یک هسته<sup>۱</sup>، مجموعه‌ای از آیتم‌های  $(\circ)$  LR برای یک گرامر است و یک گرامر  $(1)$  LR ممکن است بیش از دو مجموعه از آیتم‌ها با یک هسته تولید کند.

---

<sup>1</sup> core

## ساختن جداول تجزیه LALR

- از آنجایی که هسته  $Goto(I, X)$  فقط به هسته  $I$  بستگی دارد، توابع  $Goto$  از مجموعه‌های ادغام شده نیز ادغام می‌شوند.
- فرض کنید یک گرامر  $(1) LR$  داریم. اگر همهٔ حالت‌ها با هستهٔ یکسان را ادغام کنیم، این احتمال وجود دارد که نتیجهٔ دارای تعارض باشد اما به احتمال زیاد تعارض رخ نخواهد داد.
- می‌توان اثبات کرد که حاصل ادغام هیچ‌گاه تعارض انتقال کاهش نخواهد داشت، اما این امکان وجود دارد که تعارض کاهش رخ دهد.

## ساختن جداول تجزیه LALR

- مثال : گرامر زیر را در نظر بگیرید.

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

- این گرامر چهار رشته  $acd$  و  $ace$  و  $bcd$  و  $bce$  را تولید می‌کند. این گرامر یک گرامر LR(1) است.

- مجموعه آیتم‌های  $\{[A \rightarrow c \cdot, d], [B \rightarrow c \cdot, e]\}$  برای پیشوند پایدار  $ac$  معتبر است و مجموعه آیتم‌های  $\{[A \rightarrow c \cdot, e], [B \rightarrow c \cdot, d]\}$  برای  $bc$  معتبر است. هیچ‌کدام از این مجموعه‌ها تعارض ندارند. اما اجتماع آنها تعارض کاهش کاوش 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  $\text{LR}(1)$  را می‌سازیم و اگر تعارض ایجاد نشود هسته‌های یکسان را ادغام می‌کنیم. سپس یک جدول تجزیه از آیتم‌های ادغام شده می‌سازیم. اگر امکان ساختن چنین جدولی وجود داشت گرامر  $\text{LALR}(1)$  است.

## ساختن جداول تجزیه LALR

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

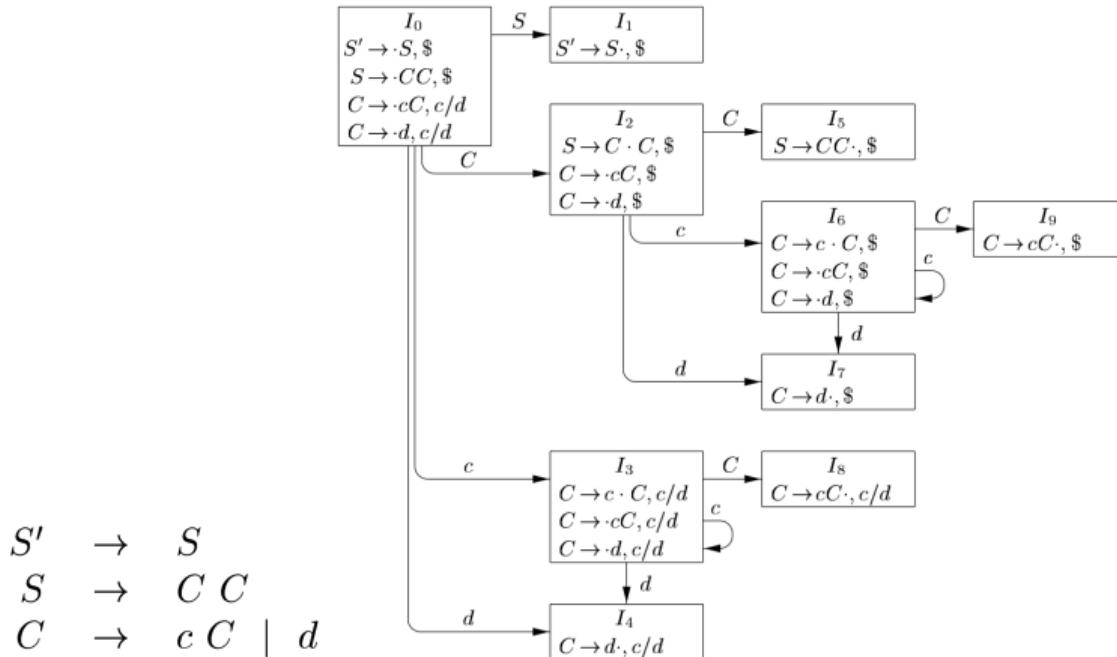
۱. گروه مجموعه آیتم‌های LR(1) را به صورت  $\{I_0, I_1, \dots, I_n\}$  می‌سازیم.
۲. برای هر هسته در بین مجموعه آیتم‌های LR(1) مجموعه‌هایی که هسته یکسان دارند را پیدا کرده و مجموعه‌های آنها را ادغام می‌کنیم.
۳. فرض کنیم مجموعه‌های آیتم‌های LR(1) به دست آمده  $\{J_0, J_1, \dots, J_m\}$  است. عملیات برای حالت  $i$  از آیتم  $I_i$  به همان صورتی که قبلاً توضیح داده شده به دست می‌آید. اگر یک تعارض وجود داشته باشد، تجزیه‌کننده LALR(1) نیست.
۴. تابع Goto به صورت زیر محاسبه می‌شود. اگر  $J$  اجتماع یک یا چند مجموعه آیتم‌های LR(1) باشد یعنی  $J = I_1 \cup I_2 \cup \dots \cup I_k$  آنگاه هسته‌های  $Goto(I_1, X)$  و  $Goto(I_2, X)$  و ... و  $Goto(I_k, X)$  یکسان هستند، زیرا  $I_1, I_2, \dots, I_k$  هسته یکسان دارند. فرض کنید  $K$  اجتماع همه مجموعه‌های آیتم‌هایی باشد که هسته آنها مانند  $Goto(I_1, X) = K$  است، آنگاه  $Goto(J, X) = K$ .

## ساختن جداول تجزیه LALR

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

# ساختن جداول تجزیه LALR

- مثال : گرامر زیر و مجموعه آیتمها و توابع گذار برای این گرامر را در نظر بگیرید.



## ساختن جداول تجزیه LALR

- در مجموعه آیتم‌ها می‌توانیم سه جفت از آیتم‌ها را ادغام کنیم:

- مجموعه آیتم‌های  $I_3$  و  $I_6$  به صورت زیر می‌توانند ادغام شوند.

$$I_{36}: \begin{aligned} C &\rightarrow c \cdot C, \quad c/d/\$ \\ C &\rightarrow \cdot cC, \quad c/d/\$ \\ C &\rightarrow \cdot d, \quad c/d/\$ \end{aligned}$$

- مجموعه آیتم‌های  $I_4$  و  $I_7$  را می‌توانیم به صورت زیر ادغام کنیم:

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

- و همچنین مجموعه آیتم‌های  $I_8$  و  $I_9$  را می‌توانیم ادغام کنیم:

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

## 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  $Goto(I_{36}, C) = I_8$  را در نظر بگیرید. در مجموعه آیتم‌های  $LR(1)$  داریم  $I_8$  و  $I_8$  عضوی از  $I_{89}$  است، بنابراین  $Goto(I_{36}, C) = I_{89}$ . از طرف دیگر اگر  $I_6$  را در نظر بگیریم به همین نتیجه می‌رسیم، زیرا  $I_6$  و  $I_9$  نیز عضوی از  $I_{89}$  است.

# ساختن جداول تجزیه LALR

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

STATE	ACTION			GOTO		STATE	ACTION			GOTO	
	<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>		<i>c</i>	<i>d</i>	\$	<i>S</i>	<i>C</i>
0	s3	s4		1	2						
1			acc								
2	s6	s7			5						
3	s3	s4			8						
4	r3	r3				0	s36	s47		1	2
5			r1			1			acc		
6	s6	s7			9	2	s36	s47			5
7			r3			36	s36	s47			89
8	r2	r2				47	r3	r3	r3		
9			r2			5			r1		
						89	r2	r2	r2		

## ساختن جداول تجزیه LALR

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

## ساختن جداول تجزیه LALR

- برای مثال، برای ورودی  $ccd\$$  ، تجزیه‌کننده CLR حالات ۰ ۳ ۳ ۴ را ببروی پشته قرار می‌دهد و در حالت ۴ یک خطا تشخیص می‌دهد. تجزیه‌کننده LALR حالات ۰ ۳ ۶ ۳ ۶ ۴ ۷ را ببروی پشته قرار می‌دهد، اما در حالت ۴۷ با ورودی  $\$$  عملیات کاهش  $d \rightarrow C$  را انجام می‌دهد و ببروی پشته ۸۹ ۰ ۳ ۶ ۳ ۶ ۰ قرار می‌گیرد. سپس یک عملیات کاهش دیگر با استفاده از  $cC \rightarrow C$  انجام داده و ۸۹ ۰ ۳ ۶ ۰ ببروی پشته قرار می‌گیرد و در نهایت با یک کاهش دیگر ۰ ۲ ببروی 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  $5^0$  تا  $10^0$  ترمینال و حدود  $10^0$  قانون تولید می‌تواند یک جدول تجزیه LALR 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 تغییر داد که همیشه تصمیم درست را در فرایند تجزیه اتخاذ کند 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 گرامرها مبهم در تجزیه‌کننده‌های LR استفاده کرد.

# بازیابی خطای در تجزیه‌کننده LR

- تجزیه‌کننده LR خطاهای را با مراجعه به جدول تجزیه شناسایی می‌کند.
- تجزیه‌کننده LR استاندارد به محض وقوع خطا، آن را شناسایی می‌کند، اما تجزیه‌کننده LALR ممکن است قبل از صدور خطا تعدادی عملیات کاهش انجام دهند.
- در تجزیه‌کننده LR بازیابی خطا با توکن همگام‌کننده<sup>۱</sup> به صورت زیر انجام می‌شود.
- پشته بررسی می‌شود تا به حالت  $s$  برسیم که با تابع  $\text{Goto}$  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  $\text{Goto}(s, A)$  برروی پشته قرار می‌گیرد و تجزیه ادامه پیدا می‌کند.

---

<sup>1</sup> panic-mode error recovery

# بازیابی خطای در تجزیه‌کننده LR

- ممکن است چند انتخاب برای متغیر  $A$  وجود داشته باشد. معمولاً 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 خطای سعی می‌کند عبارتی را که شامل خطای نحوی است حذف کند. تجزیه‌کننده تشخیص می‌دهد که رشته‌ای که از متغیر  $A$  به دست می‌آید دارای خطای خطا است. قسمتی از آن رشته پردازش شده است و نتیجه این پردازش تعدادی حالت بروی پشته است.
- 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  $Goto(s, A)$  بروی 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 خطای در تجزیه‌کننده LR

- بازیابی خطای با جایگزینی توکن‌ها<sup>۱</sup> بدین صورت پیاده‌سازی می‌شود که هریک از خانه‌های خطای در جدول تجزیه بررسی شده و تشخیص داده می‌شود چه نوع خطاهای رایج برنامه‌نویسی ممکن است در آن موقع رخداده.
- هریک از خانه‌های خطای در جدول تجزیه با اشاره‌گری به یک تابع مناسب جایگزین می‌شود. این توابع می‌توانند نمادهایی را در ورودی یا در پشته اضافه کنند و یا قسمتی از ورودی یا پشته را حذف کنند یا تغییر دهند. باید اطمینان حاصل شود که پردازش خطای باعث نمی‌شود تجزیه‌کننده وارد یک حلقه بی‌پایان شود.

---

<sup>۱</sup> phrase-level error recovery