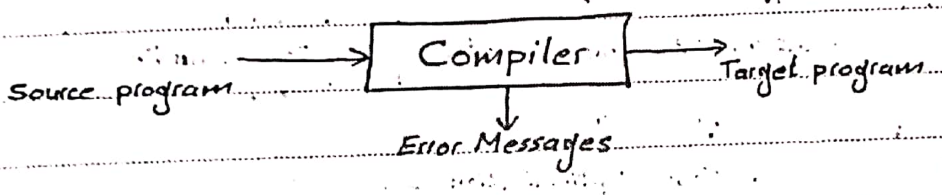
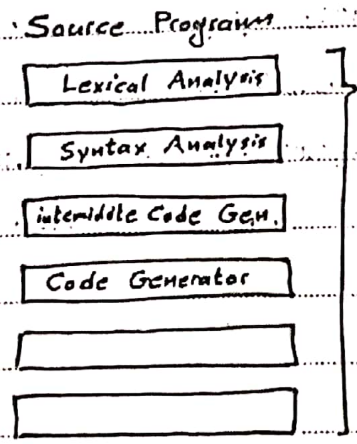
⇐ 14 نمره پایان ترم
5 نمره پروژه ← (داخل کانال تلگرام توضیح داده شده است @tnb-habibi)
1 نمره حضور غیاب

- تعریف کامپایلر ⇐ کامپایلر برنامه‌ای است که برنامه‌ی نوشته شده به زبان مبدأ را به برنامه مقصد یا (-) تبدیل می‌کند
یکی از مسئولیت‌های مهم این فرآیند گزارش دادن خطاهای برنامه‌ی مبدأ به کاربر است

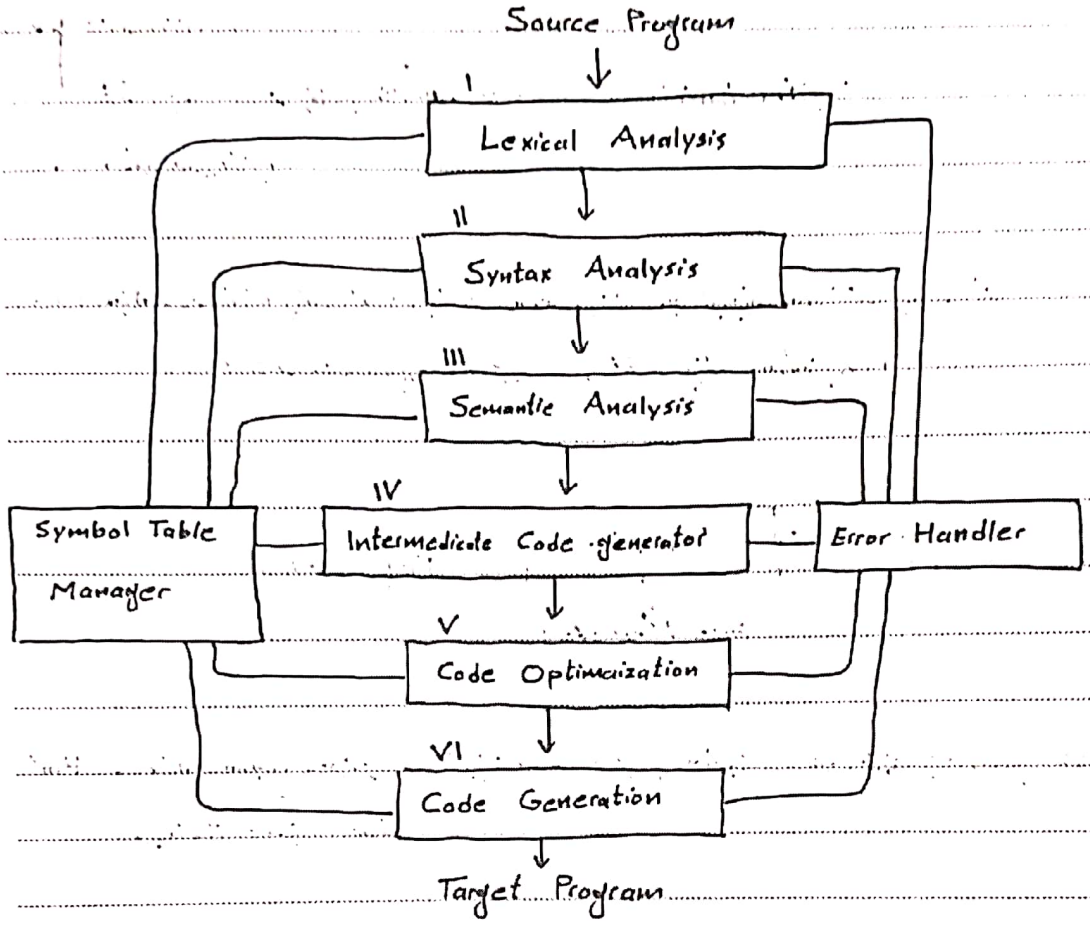


- Hint ⇐ کامپایلرها بسته به این که چگونه طراحی شده‌اند و یا با توجه به عملی که انجام می‌دهند به انواع یک گذره، چند گذره، بارگذاری و اجرا، اشکال زدایی و بهینه‌سازی دسته‌بندی می‌شوند

- مراحل کامپایل (compile)
⇐ عملیات کامپایل در شش مرحله‌ی زیر صورت می‌گیرد



⇐ این اشتباه است
(صفحه‌ی بعد)



۱- Lexical Ana... (تحلیل واژه‌ای) =>

=> وظایف مرحله کامپایلر برنامه‌ی ورودی را کاراکتر به کاراکتر خوانده و به دنبال‌های از نشانه‌ها (Token) تبدیل می‌کند

انواع مختلف نشانه‌ها عبارت‌اند از: I. کلمات کلیدی (Keyword) II. عملگرها (operations)

III. جداکننده‌ها IV. ثابت‌ها (Literals) V. شناسه‌ها (Identifiers)

Subject:
Year. Month. Date. ( )

- شناسه ها => براساس متغیرها توابع رویه ها یا (procedures) و بطور کلی هرآن چیزی که توسط کاربر انتخاب می شود شناسه گفته می شود

در اغلب زبان های برنامه نویسی کلمات کلیدی رزرو شده اند به این معنی که کاربر مجاز نیست از هیچ یک از آن ها به عنوان اسم متغیر، تابع و یا رویه استفاده نماید

- II. Syntax Ana. (تحلیل گر نحوی) =>

=> در این مرحله با استفاده از Token های تشخیص داده شده در مرحله قبل برنامه از نظر خطاهای نحوی بررسی می شود مانند تشخیص نوع متغیرها و یا تعریف نام

- III. Sematic Ana. (تحلیل معنی دار) =>

=> در این مرحله با استفاده از نتیجه تولید شده در مرحله قبل دستور از نظر معنی ای بررسی می شود.

- IV. Intomediet Code. gen (کد میانی) =>

=> یک برنامه کامپایلر برنامه ای اصلی است باید زبان میانی تولید می شود با ایجاد این کد میانی عملیات هدی که کامپایلر باید انجام دهد آسان می شود، در انتخاب زبان میانی باید موارد زیر در نظر گرفته شود. =>

=> I. تولید و پیمنه سازی که میانی باید آسان باشد
II. ترجمه آن به زبان مقصد نیز به راحتی صورت پذیرد

- V. Code Optimaization (بهینه سازی کد)

=> در این مرحله باید دستورات مجدد در کد میانی پیدا شود که باعث اجرای سریع تر برنامه گردد

- VI. Final Code Generation (تولید کد نهایی)

=> در مرحله آخر کد میانی تولید شده به کد نهایی که همان زبان مقصدی باشد تبدیل می گردد

P4PCO

Hint = فایل کامپایل شده از نوع (exe ها) می باشد و فایل اجرایی نیست

Error Handler (خطا پرداز) =

= هر بار که خطایی در یکی از مراحل Compile رخ دهد رویه ای به نام خطا پرداز فراخوان می شود این بخش سعی می کند خطا را به نحوی برطرف کند که Compiler بتواند خطاهای بیشتری را در برنامه تشخیص دهد و با اولین خطای موجود در برنامه عملیات Compile متوقف نشود.

sy.k (مدیریت جدول نشانه ها و علائم)

= یکی از کارهای مهم و اساسی کامپایلر ثبت شناسه های استفاده شده در برنامه ورودی و جمع آوری اطلاعات درباره مشخصات هر شناسه است. این اطلاعات می تواند شامل آدرس حافظه اختصاص داده شده به شناسه، نوع آن، محلی از برنامه که این شناسه در آن تعریف شده و در رابطه با رویه ها، اسم آن ها، تعداد و نوع آن ها و روشی که آن طریق آرگومان ها به رویه فرستاده می شوند مثلاً

و نوعی که رویه ها بازی گرداننده باشند { call by refrence
call by Value

= در جدول شناسه ها به ازای هر شناسه یک رکورد وجود دارد که این رکوردها شامل مشخصات شناسه ها می باشد

رکورد →

std-ID	std-nam	std-fam
١	Mary	Habibi

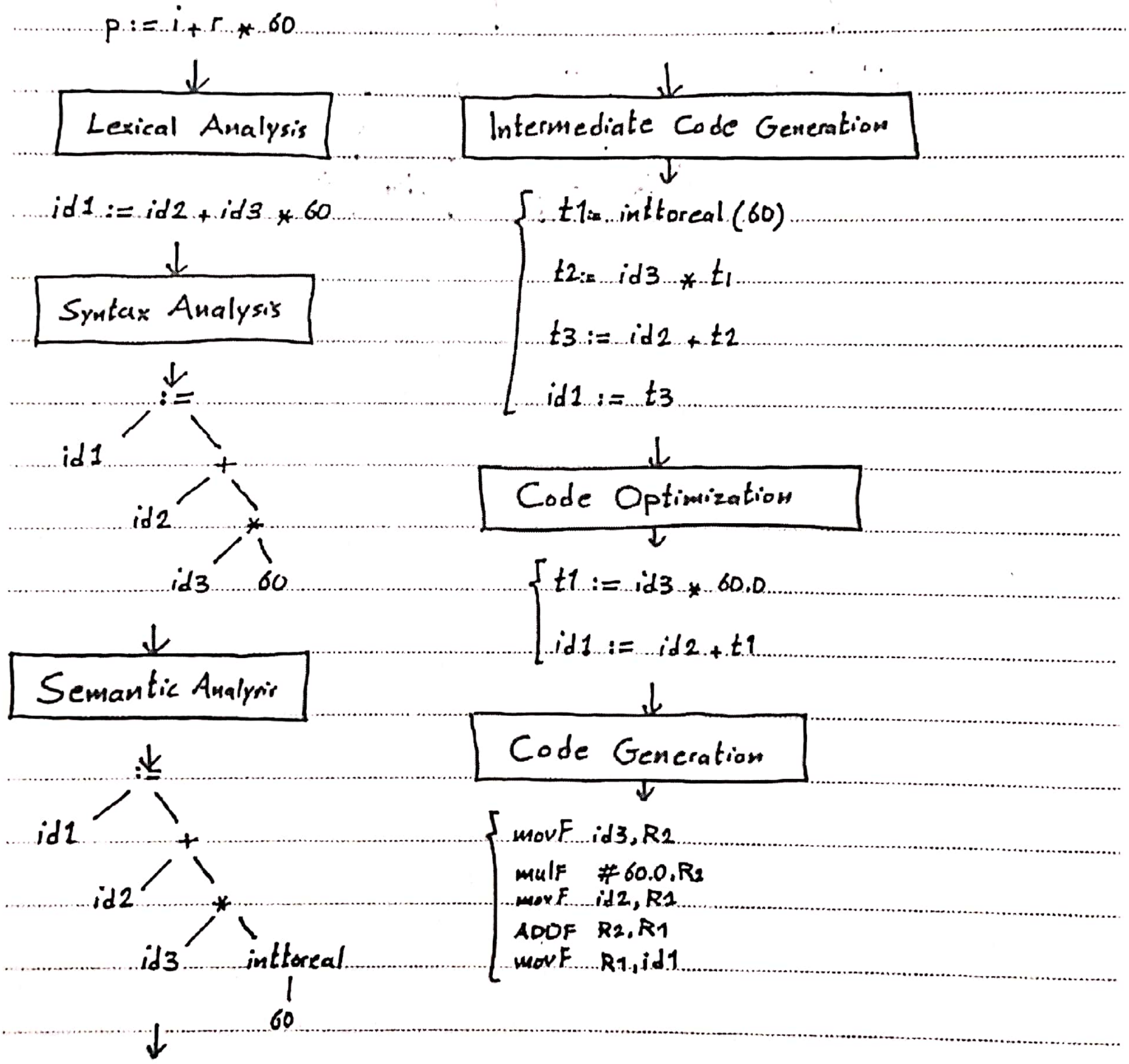
Back-End و Front-End

⇐ به چهار مرحله‌ی اول کامپایلر و بخشی از مرحله بهینه‌سازی که بستگی به زبان مبدأ دارد و مستقل از ماشین است Front-End می‌گویند

⇐ به بخشی از مرحله بهینه‌سازی و مرحله آخر Compiler که وابسته به ماشین مقصد است Back-End کامپایلر می‌گوییم

Example ⇐ شکل زیر که دستور ⇐ (position := initial + rate * 60) float ها

و تأثیر هر مرحله از Compile بر روی این دستورات نشان می‌دهد (r, i, p هر سه از نوع real هستند)

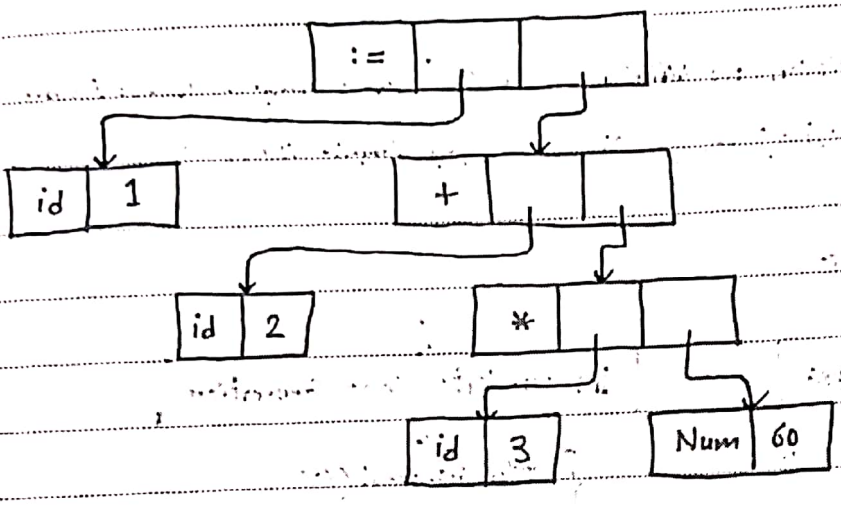


Subject:
Year. Month. Date. ( )

Hint => هر چه عمق عملگر در درخت بیتی تر باشد اولویت آن عملگر بیتی تر است

شکل زیر اولویت پردازی درخت ایجاد شده در Syntax Analy... می باشد. این درخت برگ ها 2. فیلدی هستند در حالی که

گره های میانی که شامل عملگرها است 3. فیلدی هستند چون باید هر کدام از این عملیات نیاز به دو ورودی می باشد

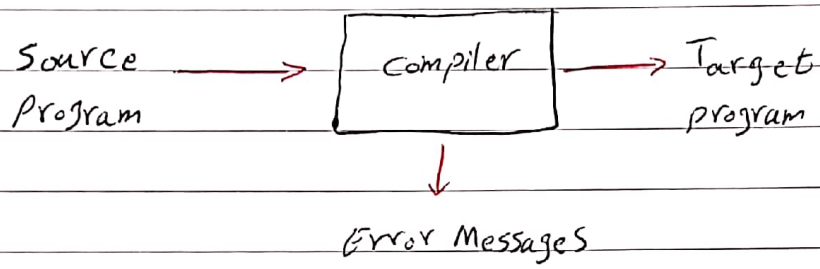


subject : کامپایلر چیس

date : ۹۷/۱۲/۱

کامپایلر:

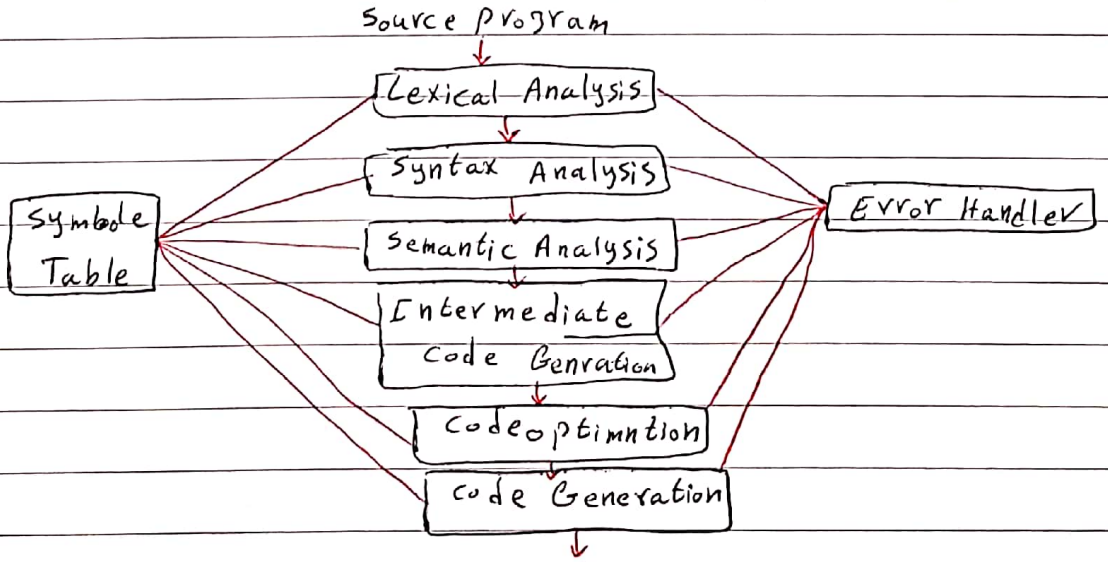
برنامه‌ای است که یک برنامه نوشته شده در یک زبان به نام زبان مبدأ یا source program را به یک زبان دیگر یعنی زبان مقصد ترجمه می‌کنیم یکی از مهمت هایی هم در این فرآیند گزارش دادن خطاهای برنامه مبدأ به کاربر است.



کامپایلرها بسیطه به گونگی طراحی و یا ترجیح به عملی که انجام می دهند به انواع یک گذره و چند گذره عبارگذاری واجرای اشکال زدایی و بهینه سازی دسته بندی می شود.

مراحل کامپایل:

عملیات کامپایل در ۶ مرحله زیر صورت می گیرد:



subject : date :

۱- تحلیل گر واژه ای (Lexical) :

برنامه ورودی را کاراکتر به کاراکتر خوانده و به دنباله ای از نشانه ها (Token) تبدیل می کند. انواع مختلف نشانه ها عبارتند از:

کلمات کلیدی (Keywords)، عملگرها (operators)، جداکننده ها (Delimiters)، ثابت ها (Literals) و

شناسه ها (IDentifiers)

- شناسه ها: به اسامی متغیرها و توابع و رویه ها (proceolures) و به طور کلی هر آنچه که توسط کاربر انتخاب می شود شناسه گفته می شود. در اغلب زبان های برنامه سازی کلمات کلیدی رزرو شده اند بدین معنی که کاربر مجاز نیست از این کلمات برای نام متغیرها و توابع و رویه ها استفاده کند.

۲- تحلیل نحوی :

در این مرحله با استفاده از نشانه های تولید شده (Token) در مرحله قبل برنامه از نظر خطاهای نحوی بررسی می شود مانند تشخیص

نوع متغیرها یا تعیین نام.

۳- تحلیل معنایی :

در این مرحله با استفاده از درخت نحو تولید شده در مرحله قبل برنامه از نظر معنایی بررسی می شود

۴- کدمیانی :

در مرحله تولید کدمیانی یک برنامه که معادل برنامه اصلی است با یک زبان میانه تولید می شود با ایجاد این کدمیانی عملیات

بعدی که کامپایلر باید انجام دهد آسان می گردد در انتخاب زبان میانی بیان موارد زیر در نظر گرفته شوند:

۱- تولید و بهینه سازی کدمیانی باید آسان باشد

subject : date :

۴- ترجمه آن به زبان مقصد باید به راحتی صورت پذیرد

۵- بهینه سازی کد میانی :

در این مرحله باید دستوراتی موجود در کد میانی به نحوی بهبود داده شود که باعث اجرای سریعتر برنامه گردد.

۶- Final code Generation

در مرحله آخر کد میانی تولید شده به کد نهایی که همان زبان مقصد می باشد (زبان ماشین) تبدیل می گردد.

نکته * فایل کامپایل شده دارای پسوند object ← .cpp می باشد و فایل اجرایی نیستند.

.o

.exe

Error Handler (خطایاب) :

هربار که خطایی در یکی از مرحله ها پیش آید رویه ای بنام Error Handler فراخوانی می شود این بخش سعی می کند خطا را به نحوی برطرف کند که کامپایلر بتواند خطاهای بیشتری را در برنامه تشخیص دهد و با اولین خطای موجود در برنامه عملیات کامپایل متوقف نگردد.

Symbol Table :

یکی از کارهای مهم و اساسی کامپایلر ثبت شناسه های استفاده شده در برنامه ورودی و جمع آوری اطلاعات درباره مشخصات هر شناسه است این مشخصات می توانند شامل حافظه ای اختصاص داده شده به شناسه، نوع آن، محلی از برنامه که این شناسه در آن تعریف شده و در رابطه با رویه ها اسم آن ها، تعداد، نوع آرگومان ها و روشی که آرگومان ها به رویه ها ارسال می شوند باشند مثلاً call by reference : call by value و نوعی که رویه ها را بازمی گرداند

subject : date :

نکته* در جدول نشانه‌ها به ازای هر شناسه یک رکورد وجود دارد که این رکوردها که شامل مشخصات شناسه‌ها می‌باشد.

Front_End / Back_End :

به چهار مرحله اول کامپایلر و بخشی از مرحله بهینه سازی که بستگی به زبان مبدأ sorce program و مستقل از ماشین است

Front_End کامپایلر می‌گویند. به بخشی از بهینه سازی و مرحله آخر که وابسته به ماشین مقصد است Back End کامپایلر می‌گویند.

مثال* شکل زیر ترجمه دستور و تأثیر هر مرحله از کامپایلر بر روی این دستور را نشان می‌دهد و i و p و r real است.

position := initial + rate * 60

p := i + r * 60

↓

Lexical Analysis

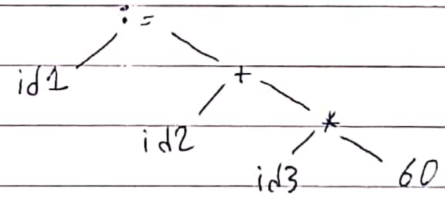
↓

id1 = id2 + id3 * 60

↓

Syntax Analysis

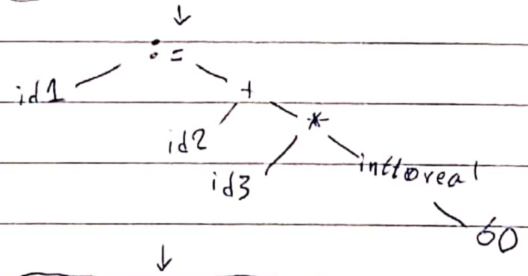
↓



↓

Semantic Analysis

↓



↓

intemediate code Generation

↓

t_1 := int to real (60)

t_2 := id3 * t_1

t_3 := id2 + t_2

id1 := t_3

↓

Code optimization

↓

t1 := id3 * 60.0

id1 = id2 + t_1

↓

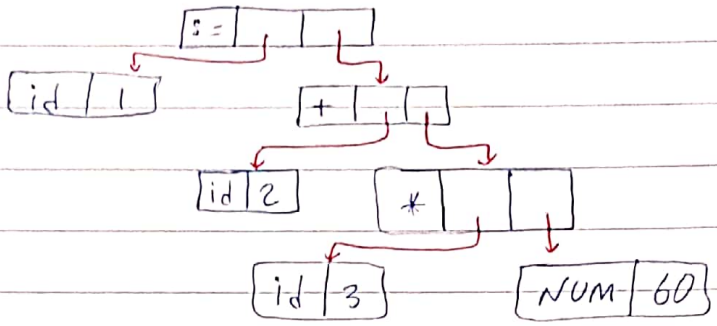
Code Geneation

↓

```
movF  id3, R2
mulF  #60.0, R2
movF  id2, R1
ADDF  R2, R1
movF  R1, id1
```

maxnote

نکته: هرچه عمق عملگر در درخت بیشتر باشد و الویت آن عملگر در عملیات بالاتر است.



در شکل بالا لیست پیوندی درخت ایجاد شده در قسمت syntax Analy می باشد این درخت برگ ها ۲ فیلدی می باشند

درحالی که گره های میانی که شامل عملگرها است ۳ فیلدی هستند چون برای هرکدام از این عملیات نیاز به دو ورودی می باشد.

- فراص عمومی زبان ها و انواع گرامرها (یا دیدگاهی نظریه زبان ها) =>

- الفبا => مجموعه ای متناهی است علائم (نمادها) یا الفبای گفته و با Σ نمایش می دهند

- رشته و یا String => دنباله ای متناهی است علائم یک الفبا را رشته می گویند $\Sigma = \{a, b\}$

$w = aaabbabababbaa$
1 2 3 4 5 6 7 8 9 10 11 12

- طول رشته => به تعداد الفبای موجود در یک رشته طول رشته می گویند و با $|w|$ نشان می دهند

$|w| = 12$

هر رشته شامل بخش هایی به صورت زیر است

- پیشوند => کنار هم قرار گرفتن یک یا چند علائم الفبا از ابتدای رشته به وجود می آید

- پسوند => از حذف یک یا چند علائم الفبا از ابتدای رشته به دست می آید

Example => رشته ی سارا را در نظر بگیرید

Example

Sara

اختیاری ✓ Sara

ara

ra

a

✓ λ

SARA

اختیاری ✓ Sara

Sar

Sa

S

✓ λ

- زیر رشته =>

=> شامل نه رشته و هر رشته و سیوندی از آن است

- زبان =>

=> مجموعه‌ای است از رشته‌های دوری یک انتهایی باشد مانند زبان لاتین

- عملیات روی زبان ها ⇐ فرض کنید L و D مجموعه هایی به صورت زیر هستند

$$\Rightarrow \begin{cases} L = \{a \dots z, A \dots Z\} \\ D = \{0, 1, 2, \dots, 9\} \end{cases}$$

⇐ آنگاه عملیات زیر را روی آن ها می توان تعریف کرد

$$\Rightarrow \begin{cases} LUD \Rightarrow \{a \dots z, A \dots Z, 0, 1, 2, \dots 9\} \\ L.D \Rightarrow \{a0, a1, a2, \dots a9, \dots, z0, z1, \dots z9, A0, A1, \dots A9, Z0, \dots Z9\} \\ L^4 \Rightarrow \{a^4, Plan, FucK, \dots\} \\ L^* \Rightarrow \{\lambda, a, \dots z, A \dots Z, a^2, z^4, A^{10}, \dots\} \\ L^+ \Rightarrow \{a \dots z, A \dots Z, a^2, z^4, A^{10}, \dots\} = L^* - \lambda \end{cases}$$

- عبارات منظم ⇐ هر عبارت منظم روی الفبا به صورت زیر تعریف می شود (هر عبارت منظم r زبان منظم L(r) را تعریف می کند)

⇐
I. λ یک عبارت منظم است که زبان منظم λ را تعریف می کند $\{\lambda\}$
II. اگر $a \in \Sigma$ باشد، a یک عبارت منظم است که زبان منظم a را تعریف می کند $\{a\}$
III. اگر r و s عبارت منظمی باشند که زبان های L(r) و L(s) را تعریف کنند آنگاه r|s عبارت منظمی است که زبان L(r) ∪ L(s) را تعریف می کند
IV. همان r.s زبان منظمی است که زبان منظم L(r).L(s) را تعریف می کند
V. همان r^* عبارت منظمی است که زبان منظم $L(r)^*$ را تعریف می کند

- Hint: دو عبارت منظم زمانی با هم معادل هستند که زبان منظم یکسانی را تعریف کنند

Ex: $a|b = b|a = b+a = a+b$

$\{a, b\} = \{b, a\}$

X

- خواص عبارات منظم =>

نوع خاصیت	EXAMPEL
۱. جابه‌جایی	$r\|s$ معادل است با $s\|r$
۱۱. شرکت‌پذیری	$(r\|s)\|t$ معادل است با $r\|(s\|t)$
	$(r.s).t$ معادل است با $r.(s.t)$
۱۱۱. توزیع‌پذیری	$(r.t)\|(s.t)$ معادل است با $(r\|s)t$
۱۷. عنصر خنثی	$s.\lambda = \lambda.s = s$

- گرامر => شامل مجموعه‌ای است از قواعد است که ساخت جملات یک زبان را مشخص می‌کند و در واقع گرامر به ما کمک می‌کند تشخیص دهیم آیا جمله‌ای متعلق به یک زبان است یا خیر

- Hint => هر گرامر یک زبان را با چهارتایی $G = (S, N, T, P)$ توصیف می‌کند

S: start, N: non-terminal, T: terminal, P: protocol

=> $S \in N$

اگر F داشتیم $F \subseteq N$

- Example => یک مثال از گرامر =>

$N = \{E\}$

$T = \{(,), *, +, id\}$

$S = \{E\}$

$$P: \{E \to E*E,\ E \to E+E,\ E \to (E),\ E \to id\}$$

- زبان گرامر => زبانی که توسط گرامر تعیین می شود به صورت زیر بیان می گردد

$$\Rightarrow L(G) = \{\alpha \mid \beta \overset{+}{\Rightarrow} \alpha,\ \alpha \in T^*,\ \beta \in (TUN)^+ - \varepsilon\}$$

- Hint => β, α رشته هایی شامل ترکیبی از T, N می باشد یعنی زبان $L(G)$ شامل رشته هایی است که اگر از علامت شروع یعنی S آغاز کنیم با استفاده از قواعد موجود در گرامر به یک یا چند فرم رشته ای α را تولید نموده که در آن فقط شامل λ و هر ترکیبی از terminal ها است

- انواع گرامر => گرامر چامسکی => چامسکی با در نظر گرفتن محدودیت هایی روی گرامرها آن ها را به چهار دسته تقسیم می کند =>

I. گرامر نوع صفر یا بدون محدودیت
II. گرامر نوع یک یا با محدودیت
III. گرامر نوع دو یا مستقل از متن
IV. گرامر نوع سه یا منظم

I. گرامر نوع صفر یا بدون محدودیت =>

=> این گرامر شامل قاعده ای به صورت $\alpha \rightarrow \beta$ می باشد که محدودیت خاصی ندارد فقط شرط این است که در این قاعده یعنی α باید حداقل شامل یک Non-Terminal باشد

II. گرامر نوع یک یا با محدودیت =>

=> این گرامر علاوه بر محدودیت که در شرط قبل گفته شد باید در آن طول رشته سمت چپ قاعده => طول رشته سمت راست قاعده باشد

$$(|\alpha| \leq |\beta|)$$

Example => $S \rightarrow aAb$ ✓ با محدودیت

$aA \rightarrow aaA$ ✓ با محدودیت

$abaA \rightarrow aB$ X بدون محدودیت

۱۱۱. گرامر نوع دو یا مستقل از متن =>

=> علاوه بر محدودیت قبلی حتی باید سمت چپ قاعده تنها از یک Non-Terminal باشد

۱۷. گرامر نوع سه یا منظم =>

=> علاوه بر تمام محدودیت هایی که در بالا گفته شد گرامر نباید شامل λ باشد

- انواع بسط => با شروع از علامت Start و جایگذاری Non-Terminal ها با قاعده موجود آن ها به یک گرامری تبدیل یک سری از کلمات را تولید کرد که به آن ها بسط آن گرامر می گویند

- انواع بسط => دو نوع بسط وجود دارد بسط چپ که در آن همواره چپ ترین Non-Terminal جایگذاری می شود

=> بسط راست که در آن همواره راست ترین Non-Terminal با استثنا با توجه موجود در گرامر جایگزین می شود

- Example => گرامر زیر را در نظر بگیرید برای کلمه (id+id*id) بسط چپ و راست بنویسید

=>
- $E \rightarrow E+E$
- $E \rightarrow E*E$
- $E \rightarrow (E)$
- $E \rightarrow id$

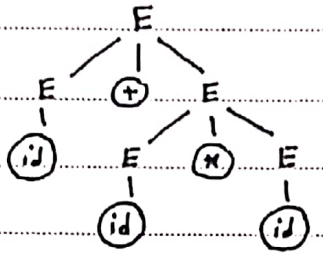
بسط راست => $E \rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*id \Rightarrow E+id*id$

بسط چپ => $E \rightarrow E+E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E$

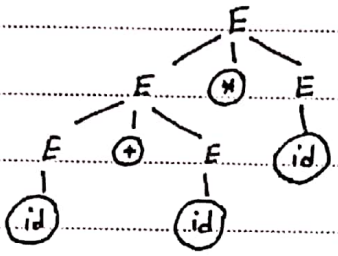
$id+id*id$

PqPCO

- کشیدن درخت های آن ها =>



سبط راست =>



سبط چپ =>

- Example => گرامری بنویسید که زبان $L(G) = \{a^n b^n, n \geq 0\}$ را تولید کند

جواب => $S \rightarrow aSb \mid \lambda$

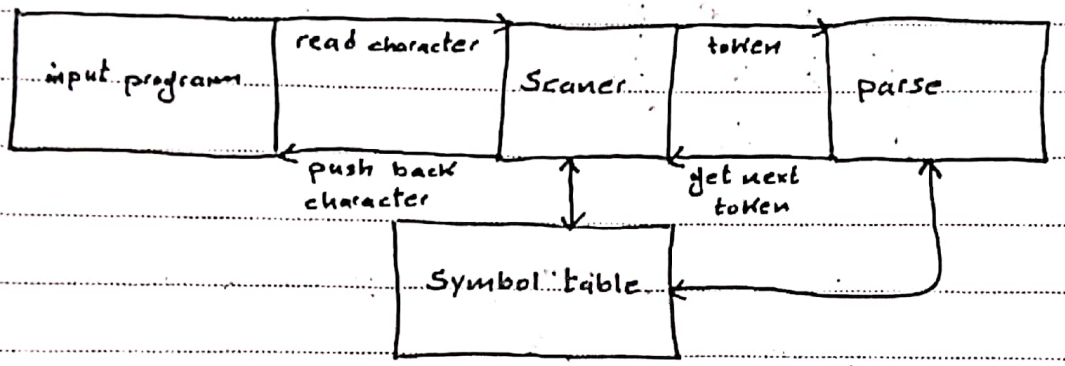
- Example => گرامر $S \rightarrow abS \mid \lambda$ چه زبانی تولید می‌کند $L(G)$

جواب => $L(G) = \{(ab)^n, n \geq 0\}$

- Example => گرامری بنویسید که زبان $L(G) = \{a^n b^{2n}, n \geq 0\}$ را تولید کند

- تحلیل گر لغوی (Lexical Analysis)

⇐ تحلیل گر لغوی اولین فاز compiler را تشکیل می دهد و وظیفه اصلی آن خواندن کاراکترهای ورودی و تولید دنباله ای از نشانه ها (tokens) است که تجزیه کننده برای تحلیل لغوی با آن استاندارد می کند. ارتباط به شکل زیر نشان داده شده است و معمولاً با فراخوانی تحلیل گر لغوی به عنوان یک زیر روال یا "subroutine" یا هم روال یا "coroutine" از تجزیه کننده پیاده سازی می شود. به محض دریافت توسط "نشانه بعدی را بگیر" از تجزیه کننده کاراکترهای ورودی را تا مشخص شدن نشانه بعدی می خواند معمولاً به تحلیل گر لغوی "Scaner" و به تجزیه کننده "parser" می گویند



(پشتی)

- Example ⇐ اگر فرمان ورودی زبان "Fortran" "E = M * C ** 2" باشد "token" های شناخته داده شده برای این زبان

⇒ به صورت زیر است

<id, pointer to symbol-table Entry for E>
<assign_op>
<id, pointer to symbol_table entry for M>
<mult_op>
<id, pointer to symbol_table Entry for C>
<exp_op>
<num, integer value 2>

- Hint <= scanner یعنی برای شناسه‌ها با تشخیص می‌دهد آن‌ها را باید در جدول نمادها (symbol table) برای "parser" می‌فرستد ، scanner همچنین می‌تواند فضاهای خالی و توضیحات "comment" را نیز از متن ورودی حذف کند

- Token, Pattern, Lexeme <=

<= جدول زیر مجموعه‌ای از الگوها و لغات مربوط به "Token" را نشان می‌دهد

Token	Sample Lexeme	Informal Descreption of Pattern
const	Const	Const
if	if	if
relation	< , = , <> , > , >= , <=	< or <= or = or <> or > or >=
id	pi , count , D2	letter followed by letters and digits
num	3.142 , 0 , 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " expect "

- Hint ⇐ به طور کلی ورودی مجموعه‌ای از رشته‌ها وجود دارند که برای آن‌ها توکن (نشانه‌)های مشابه تولید می‌شود. این مجموعه از رشته‌ها با قانونی به نام الگوی مربوط به نشانه توصیف می‌شود. این الگو با هر رشته در مجموعه منطبق است. یک لغت مثال‌هایی از جنبه‌های الگو در برنامه‌ی مبدا است که ساختارها با الگوی یک نشانه است. به مثال‌هایی از کاراکترها که تشکیل یک token می‌دهند واژه یا لغت یا Lexeme آن token می‌گویند.

- Hint ⇐ در بعضی از وضعیت‌ها Scanner قبل از اینکه تصمیم بگیرد چه token را به parser بفرستد نیاز دارد که چند کاراکتر دیگر را نیز از ورودی بخواند. به عنوان مثال Scanner با دیدن علامت "<" در ورودی باید کاراکتر بعدی را نیز بخواند در صورتی که این کاراکتر "=" باشد token "<=" و در غیر این صورت token "<" کوچکتر تشخیص داده می‌شود.

- Hint ⇐ در زبان‌های ساده فضای خالی (کاراکتر فاصله) به جز در رشته نادیده گرفته می‌شود (فضای خالی). به عنوان مثال اگر کلمه‌ی "do" را در نظر بگیریم "Do 5 I = 1.25" تا زمانی که Scanner به علامت "," نرسیده باشد نمی‌توان گفت "Do" کلمه‌ی کلیدی یا بخشی از شناسه‌ی "Do5I" می‌باشد.

⇒ { Do 25
5
I
=
1
,
25

- Hint ⇐ در برخی از زبان‌ها بعضی از رشته‌ها به عنوان کلمات کلیدی رزرو شده است یعنی کاربر نمی‌تواند از کلمات کلیدی به عنوان نام متغیرها، توابع و... استفاده نماید یا تغییر دهد. اگر کلمات کلیدی رزرو نشده باشند آنگاه تحلیل‌گر لغوی باید بین کلمه‌ی کلیدی و شناسه‌ی تعریف شده توسط کاربر تمایز قائل شود. به عنوان مثال در زبان برنامه‌نویسی PL/I کلمات کلیدی رزرو نشده‌اند بنابراین تشخیص کلمات کلیدی از شناسه‌ها دشوار است. معمولاً توسط Parser تشخیص داده می‌شود [illegible]

⇒ { IF then THEN then = else;
ELSE else = then;

- Hint ⇐ تشخیص کلمه‌ی کلیدی THEN از متغیر then بسیار مشکل است (با توجه به اینکه تا انتهای خط کوچک...) بسیار مشکل است. در این موارد معمولاً parser هم تشخیص نهایی را می‌دهد.

- خطاهای واژه‌ای: ⇐

⇐ به طور کلی خطاهای محدودی را scanner می‌تواند تشخیص دهد زیرا scanner تمام برنامه‌ی ورودی را یکجا نمی‌تواند ببیند بلکه هربار قسمتی کوچکی از برنامه‌ی ورودی را می‌بیند به عنوان مثال هرگاه رشته‌ی ورودی fi در یک برنامه‌ی C برای اولین بار به صورت زیر ظاهر شود scanner قادر نیست تشخیص دهد که آیا fi یک املای غلط از کلمه‌ی کلیدی if است یا یک شناسه‌ی تابعی اعلام نشده.

fi ⇐ جایگزینی (با فرگذاری) ورودی

fi (a := a f(x))

- Hint ⇐ scanner، token fi را به عنوان یک شناسه به parser می‌فرستد تا این که parser درباره‌ی مورد تصمیم بگیرد اما ممکن است خطاهایی پیش بیاید که scanner قادر به انجام هیچ عملی نباشد. در این حالت برنامه‌ی خطاپرداز "Error Handler" صدا فراخوانی می‌شود تا آن خطا را به نحوی برطرف کند.

⇐ روش‌های مختلفی برای این کار وجود دارد که ساده‌ترین آن‌ها روشی معروف به "panic mode" (تحلیلی) است.

⇐ در این روش آن قدر از رشته‌ی ورودی حذف می‌شود تا این که یک "token" قابل قبول تشخیص داده شود.

⇐ سایر روش‌های تصحیح خطا عبارت اند از:

I. حذف یک کاراکتر اضافی (:= ⟶ :$=)

II. درج کاراکتر حذف شده (:= ⟶ :)

III. جایگزینی یک کاراکتر درست با یک کاراکتر نادرست (:= ⟶ :::)

IV. جابه‌جایی دو کاراکتر مجاور (:= ⟶ =: , if ⟶ fi)

- روش‌های جهت بهبود کار Scaner =>

۱. میانگیری (باندگذاری) ورودی

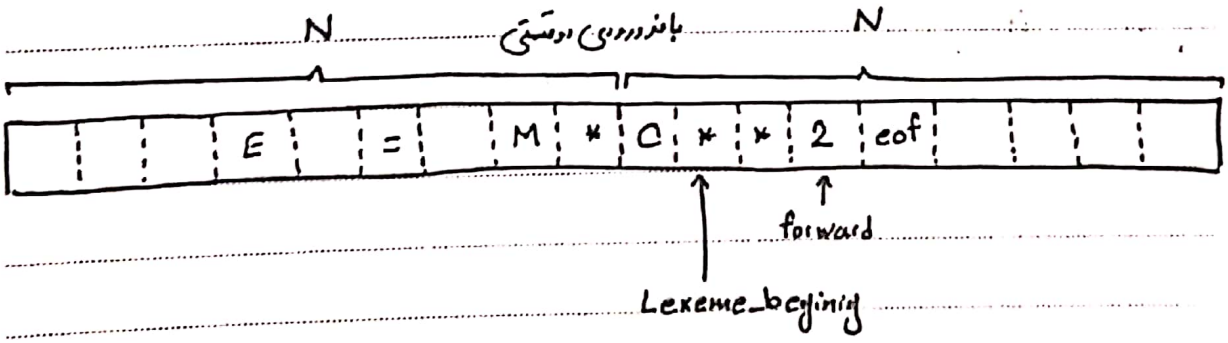
در بسیاری از مواقع scaner برای تشخیص نهایی lexem ها و متعلقات بان آن با الگوهای موجود نیاز دارد که تعداد بیش تری از کاراکترهای ورودی را بخواند. واضح است که مقدار زیادی از زمان مربوط به کاراکترها صرف می‌شود.

برای جلوگیری از این امر می‌توان از تکنیک‌های پیشنهاد شده از باند بهره جست. در این تست باندهای روش‌های مفید برای افزایش سرعت تحلیل گرهای ساخته شده از نگهبان برای مشخص کردن باند آشنا شویم.

۱. بافت میانگین گیری

=> در بسیاری از زبان‌های مبدأ تحلیل گرهای قبل از پیدا کردن یک سرنگاری باید بخشی کاراکتر بعد از لغت نگاه کنند تا بتوانند الگوی مورد نیاز برای شناسایی آن لغت را پیدا کنند.

تحلیل گرهای برای این منظور از یک تابع برای برگرداندن کاراکترهای پیش‌نگر (look ahead) استفاده می‌کنند از آن جا که ممکن است زمان زیادی صرف انتقال کاراکترها شود. از این رو از روش‌های باندگیری استفاده می‌شود تا سربار مورد نیاز برای بدست آوردن کاراکترهای ورودی کاهش یابد. برای روش از یک باند که حالت شکل زیر به دو قسمت N کاراکتری تقسیم شده است استفاده می‌کنیم که معمولاً N تعداد کاراکتر موجود ورودی یک Block از دیسک است.



Hint ← به این ترتیب به جای فراخوانی دستور Read برای هر کاراکتر می‌توان دستور Read را برای خواندن N کاراکتر به کار برد و آن‌ها را در هر نیمه بافر قرار داد. اگر کمتر از N کاراکتر ورودی باقی مانده باشد آنگاه بعد از آخرین کاراکترهای ورودی یک کاراکتر مخصوص بنام "eof" ← (end of file) داخل بافر قرار می‌گیرد

Hint ← eof پایان فایل مبدأ را مشخص می‌کند و با کاراکترهای ورودی متفاوت است.

Hint ← در این روش از دو اشاره‌گر استفاده می‌شود و رشته‌ی کاراکتری بین این دو اشاره‌گر لغت جاری می‌باشد

در آغاز هر دو اشاره‌گر به اولین کاراکتر لغت بعدی پیدا نشده اشاره می‌کنند. اشاره‌گر forward تا رسیدن به token مورد نظر پیش می‌رود

Hint ← به محض آن که لغت بعدی تعیین شد اشاره‌گر forward طوری تنظیم می‌شود تا به انتهای کاراکتر سمت راست این لغت اشاره کند :

بعد از پردازش لغت هر دو اشاره‌گر طوری تنظیم می‌شوند تا به کاراکتری که بلافاصله بعد از این لغت قرار دارد اشاره کنند

بدین ترتیب توضیحات و کاراکترهای فضای خالی را می‌توان به صورت الگوهایی در نظر گرفت که هیچ token مشابهی ندارند

Hint ← اگر اشاره‌گر forward از نیمه بافر بگذرد نیمه‌ی چپ بافر با N کاراکتر بارگذاری می‌شود و اشاره‌گر forward به ابتدای بافر برمی‌گردد و به آدرس شروع نیمه‌ی اول بافر اشاره می‌کند

Hint ← این روش استفاده از بافر در بیشتر موارد کاملاً خوب عمل می‌کند ولی در مواقعی که برای تشخیص "token" نیاز به خواندن کاراکترهایی

• بیش از طول بافر باشد به دلیل محدودیت Look Ahead درست عمل نمی‌کند

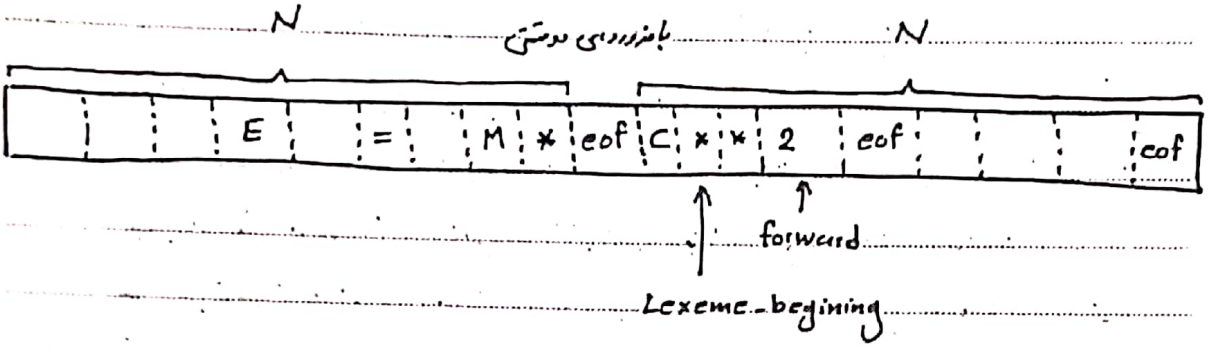
برای کنترل حرکت اشاره‌گر forward می‌توان دستورات زیر را بیان کرد.

```
if forward at end of first half then begin
    reload second half;
    forward := forward + 1;
end
else if forward at end of second half then begin
    reload first half;
    move forward to begining of first half;
end
else forward := forward + 1;
```

Hint: [سودو] کد مربوط به حرکت اشاره‌گر forward

۱۱- استفاده از نگهبان (Sentinel) =>

=> در روش قبل در هر بار دریافت "forward" باید چک شود که آیا به انتهای هر یک از دو نیمه بافر رسیده است یا خیر
یعنی در هر مرحله مقایسه با یه انجام شود حال اگر هر انتهای نیمه بافر را با استفاده از کاراکتر مخصوص "eof" بعنوان
کاراکتر نگهبان مشخص کنیم این دو مقایسه به یک مقایسه کاهش می یابد



=> نگهبان کاراکتر خاصی است که نمی تواند جزئی از برنامه مبدأ باشد

```
forward := forward + 1;
if forward ↑ = eof then begin
    if forward at end of first half then begin
        reload second half;
        forward := forward + 1;
    end
    else if forward at end of second half then begin
        reload first half;
        move forward to beginning of first half;
    end
    else /* eof within a buffer signifying end of input */
        terminate lexical analysis
    end
```

P4PCO

دیاگرام های انتقال یا نمودارهای تبدیل ⇐

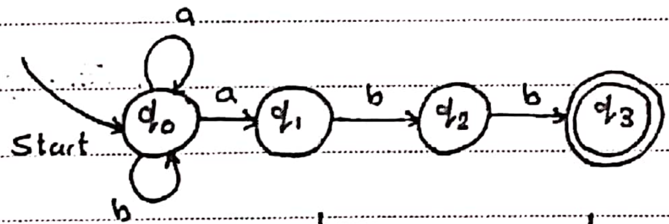
⇐ برای پیاده سازی دستی یک Scaner از ابزاری بنام دیاگرام انتقال یا نمودار تغییر حالت کمک می گیریم به این معنی که به عنوان یک مرحله میانی در ساخت تحلیل گر لغوی ابتدا نمودار را ترسیم می کنیم که به آن نمودار تغییر حالت می گویند یک دیاگرام انتقال در واقع یک گراف جهت دار است که هر یک از گره های آن معرف یک وضعیت یا "State" است.

یکی از وضعیت ها به عنوان وضعیت شروع و یک یا چند تا از آن ها به عنوان وضعیت خاتمه مشخص می شود.

برچسب یا Label های روی لبه های یک دیاگرام انتقال قرار داده می شود که مشخص می کند در چه صورتی می توان از این وضعیت به وضعیت دیگر رفت. هر دیاگرام انتقال معرف یک زبان است با دنبال کردن کاراکترهای یک رشته و تطبیق آن ها با برچسب های دیاگرام انتقال معرف یک زبان و پیمایش آن دیاگرام می توان مشخص نمود که آیا آن رشته متعلق به زبان مورد نظر است یا خیر.

فرض می کنیم نمودارهای تبدیل این بخش معین و قطعی هستند (DFA) ⇐ یعنی یک نماد نمی تواند متعلق به برچسب های دو یال باشد که از یک حالت خارج می شود.

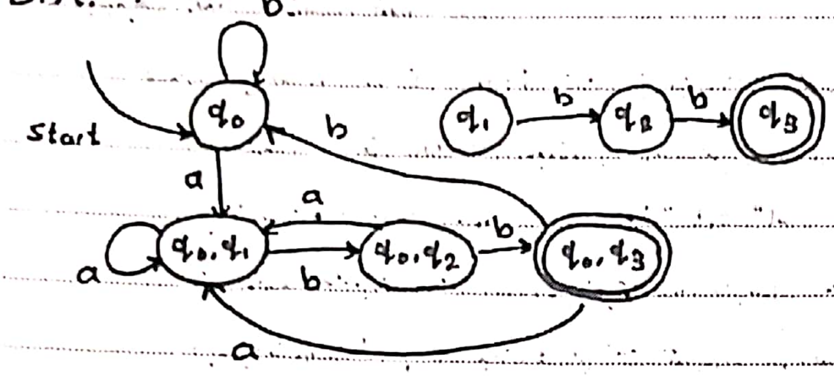
Example ⇐ دیاگرام انتقال زیر عبارت منظم "$(a|b)^*abb$" هر دو زبانی را توصیف می کند که شامل رشته های تشکیل شده از علائم a و b هستند که به "abb" ختم می شوند.



NFA: غیرقطعی

	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	—	$\{q_2\}$
q_2	—	$\{q_3\}$
(q_3)	—	—
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
$\{q_0, q_3\}$	$\{q_0, q_1\}$	$\{q_0\}$

DFA:

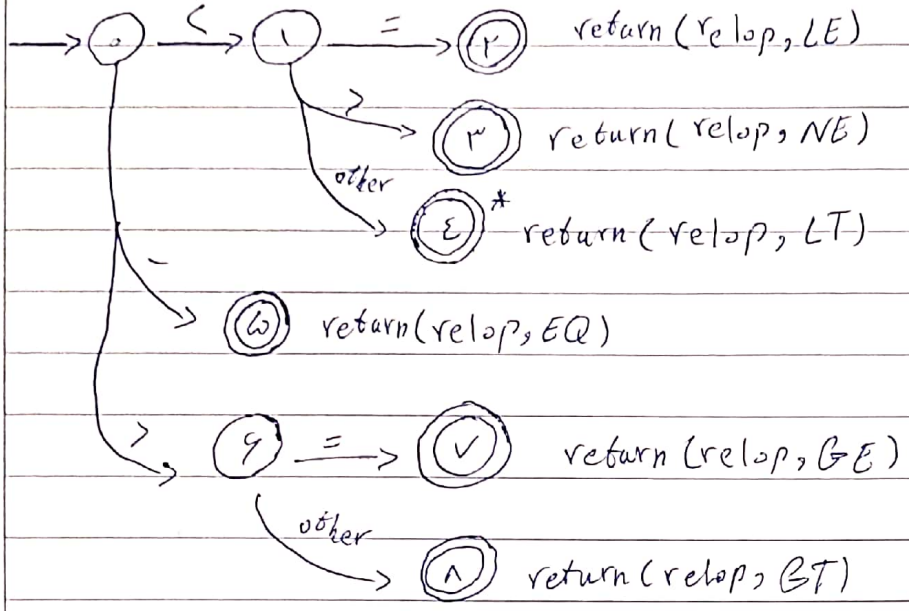


Hint ⇐ بنابراین برای پیاده سازی یک "Scaner" ابتدا دیاگرام های انتقال حروف الگوی token های زبان مورد نظر رسم می گردد. این دیاگرام ها باید به صورت "DFA" باشند تا کار تحلیل گر لغوی آسان شود.

Hint ⇐ "DFA" ها برای به دست آوردن اطلاعات در مورد کاراکترهایی که به وسیله اشاره گر "forward" مرور می شوند استفاده می گردند. بدین ترتیب که همان طور که کاراکترهای ورودی خوانده می شوند از یک وضعیت در دیاگرام به وضعیت دیگر حرکت می کنیم تا این که به یک وضعیت نهایی برسیم.

Example ⇐

مثال* یک دیاگرام انتقال برای Token های رابطه‌ای رسم کنید. (<, <=, =, <>, >, >=)



relop → < | <= | = | <> | > | >=

*other یعنی کاراکترهایی که از state قبل خارج شدن (شامل space ، عدد و حرف)

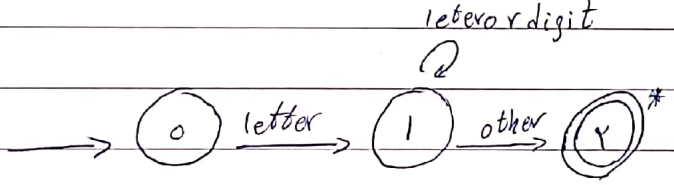
*علامت star (*) یعنی کاراکتر خوانده شده جزء علایم مقایسه نیست و باید به ورودی برگردد

states	return	
(0,1,2)	(relop, LE)	کوچکتر مساوی
(0,1,3)	(relop, NE)	نابرابر
(0,1,4)	(relop, LT)	کوچکتر
(0,5)	(relop, EQ)	برابر
(0,6,7)	(relop, GE)	بزرگتر مساوی
(0,6,8)	(relop, GT)	بزرگتر

subject : date :

به طور کلی ممکن است چند نمودار تبدیل وجود داشته باشد که هرکدام یک گره از نشانه ها را مشخص می کند اگر هنگام دنبال کردن نمودار تبدیل خطایی اتفاق بیافتد در این صورت اشاره گر Forward را به حالت شروع این نمودار می بریم و نمودار تبدیل بعدی را فعال می کنیم از آنجایی که اشاره گر شروع کننده رفت و اشاره گر Forward در حالت شروع نمودار به یک مکان اشاره می کنند از این رو اشاره گر Forward به مکانی برمی گردد که توسط اشاره گر شروع رفت مشخص شده است اگر در نمودارهای تبدیل خطایی آشکار نمود در این صورت خطای لغوی اتفاق افتاده و روال تصحیح خطا احضار می شود

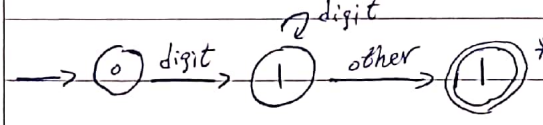
مثال * از آن جا که کلمه های کلیدی دنباله ای از حروف هستند اما از این قانون که دنباله ای از حروف و ارقام که با استفاده از حرف شروع می شوند شناسه است مستثناست بجا رنشان دادن حروف و ارقام در حالت های استثناء یا نمودارهای تبدیل روش صفیه آن است که کلمه های کلیدی را به صورت شناسه های خاص مورد بررسی قرار دهیم بدین ترتیب دیاگرام انتقال شناسه ها (مانند نام متغیرها) به صورت زیر است:



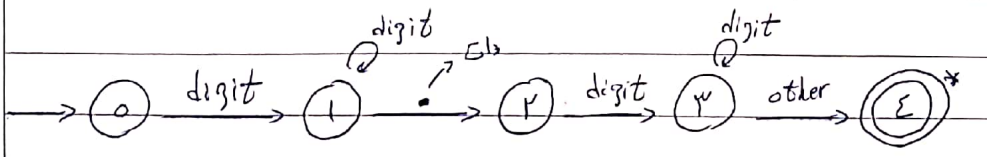
id → letter (letter | digit)*

مثال * دیاگرام انتقال اعداد صحیح و سپس اعداد نرمال بدون علامت را رسم کنید.

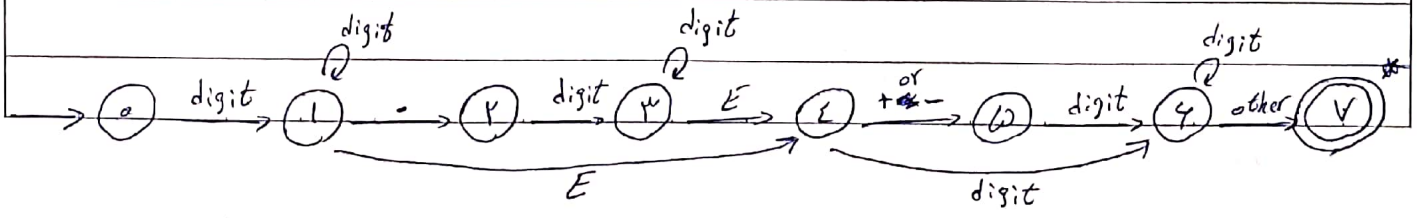
num → digit⁺ (. digit⁺)? (E (+|-)? digit⁺)?



اعداد صحیح



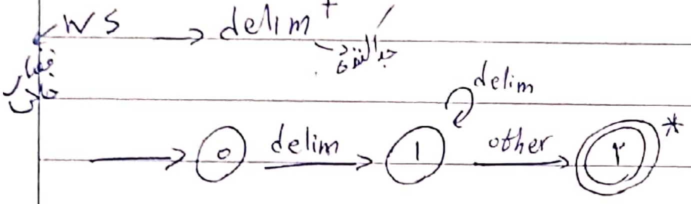
اعداد اعشاری



مثال * فرض می‌کنیم لغت‌ها بوسیله‌ی فضای سفید متشکل از دنباله‌های غیر تهی از فضاهای خالی، Tab ها و عملگر New line است از یکدیگر جدا شده اند. تحلیل‌گر لغوی باید بتواند فضاهای سفید را حذف کند. دیاگرام انتقال فضاهای خالی را رسم کنید.

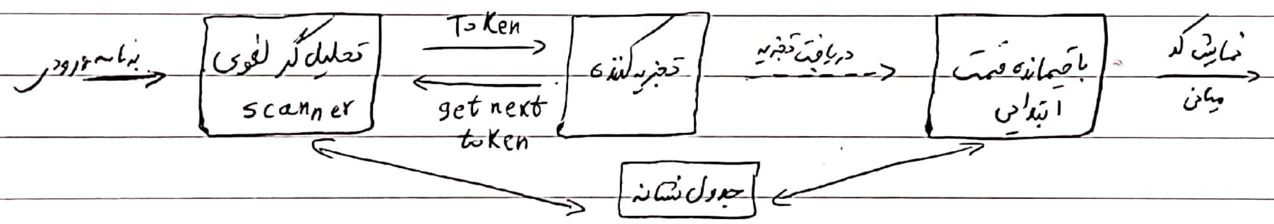
delim → space | Tab | new line

ws → delim+



* تحلیلگر نحوی :

در مرحله‌ی تحلیلگر نحوی برنامه از لحاظ دستوری مورد بررسی قرار می‌گیرد. parser در واقع دنباله‌ای از Token ها را از اسکنر دریافت کرده و بررسی می‌کند که آیا رشته ورودی با استفاده از گرامرهای زبان قابل تولید است یا خیر.



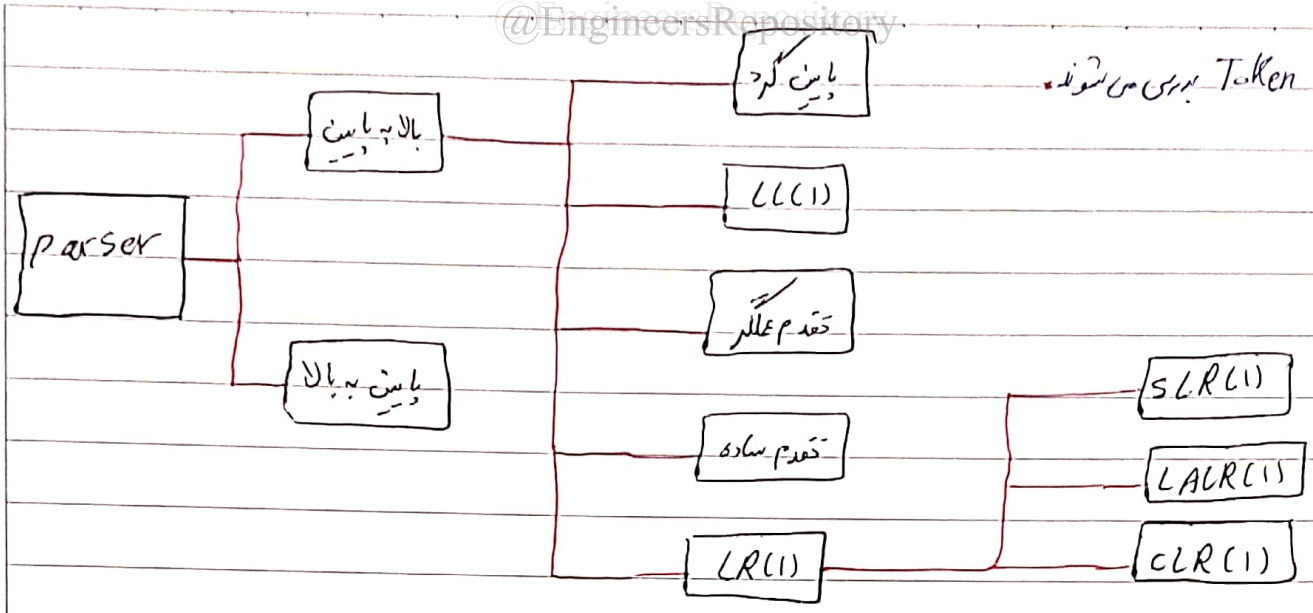
پارسر با استفاده از نشانه های موجود در جدول نشانه درخت پارس را ایجاد می‌کند. روش هایی که معمولا در کامپایلرها مورد استفاده قرار می‌گیرند به دو دسته‌ی کلی زیر تقسیم می‌شوند :

۱. بالا به پایین (Top down).

۲. پایین به بالا (bottom up).

در روش بالا به پایین درخت تجزیه از بالا یا ریشه به پایین یا برگ ها ایجاد می‌شود حال آنکه تجزیه کننده‌های پایین به بالا از برگ ها شروع می‌کنند و به طرف ریشه پیش می‌روند. در هر دو روش رشته‌ی ورودی از چپ به راست بررسی می‌شود و در هر قدم

فقط یک Token بررسی می‌شود.

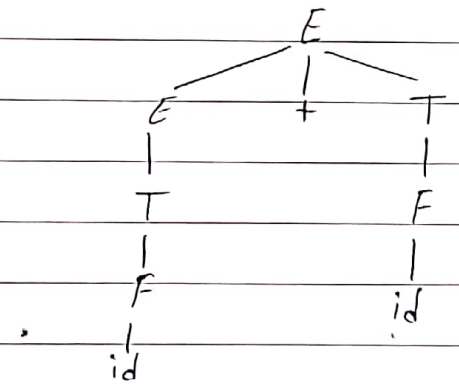


مثال: گرامر زیر را در نظر بگیرید درخت تجزیه id+id را به صورت بالا به پایین رسم کنید.

$E \rightarrow E+T \mid T$

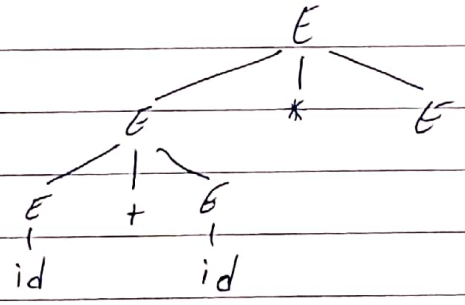
$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$



مثال: برای گرامر زیر یک درخت اشتقاق بالا به پایین به منظور id+idxid رسم کنید.

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



صحیح است

* ابهام:

به گرامری مبهم می گویند که برای یک رشته بیش از یک درخت تجزیه دارد.

* رفع ابهام:

گاهی اوقات یک گرامر را می توان به منظور رفع ابهام آن بازنویسی کرد. برای این منظور از روش های زیر استفاده می کنیم:

۱- حذف بازگشتی چپ:

یک گرامر بازگشتی چپ نامیده می شود که دارای اشتقاقی به صورت $A \overset{+}{\Rightarrow} A\alpha$ باشد. روش های تجزیی بالا به پایین نمی توانند گرامرهای بازگشتی چپ را مورد بررسی قرار دهند از این رو باید بتوان بازگشتی چپ را رفع کرد. برای این منظور به جای قوانین بازگشتی چپ $A \to A\alpha \mid \beta$ می توان قوانین غیر بازگشتی چپ را قرار داد.

$$\begin{cases} A \to \beta A' \\ A' \to \alpha A' \mid \lambda \end{cases}$$

مثال: گرامر زیر را برای عبارات محاسباتی در نظر بگیرید و سپس قوانین بازگشتی چپ مستقیم را حذف کنید.

α β

$E \to E+T \mid T$ $\quad$ $E \to TE'$

$E' \to +TE' \mid \lambda$

$T \to T*F \mid F$ $\quad$ $T \to FT'$

$T' \to *FT' \mid \lambda$

$F \to (E) \mid id$ $\quad$ $F \to (E) \mid id$

۲- فاکتورگیری از چپ:

فاکتورگیری از چپ یک تبدیل گرامر است که برای تولید گرامر مناسب برای تجزیه کننده‌های پیشگو مفید است. در حالت کلی اگر $A \to \alpha\beta_1 \mid \alpha\beta_2$ می توان قوانین تولید A را تغییر داد به این دلیل که اگر دو قانون تولید A و ورودی با یک رشته

غیر بتی دستوری شدن A شروع شود در آن صورت می دانیم که آیا A را با $\alpha\beta_1$ یا $\alpha\beta_2$ تعمیم دهیم برای رفع این مشکل از فاکتورگیری چپ به صورت زیر استفاده می‌کنیم:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

مثال: گرامر زیر مسئله else سرگردان را نشان می دهد.

$S \rightarrow iEtS | iEtSeS | a$

$E \rightarrow b$

در اینجا if و then و else به معنی i و t و e و E و S به معنی Expression (دستور)، و Statement (عبارت)، این گرامر

فاکتورگیری چپ به صورت زیر خواهد بود:

$S \rightarrow iEtSS' | a$

$S' \rightarrow \lambda | eS$

$E \rightarrow b$

*تجزیه بالا به پایین

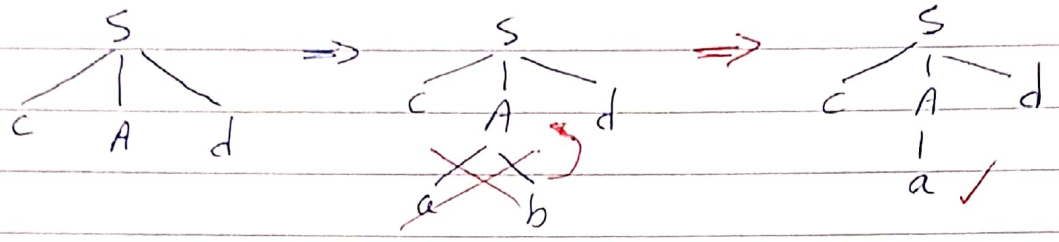
تجزیه کننده های بازگشتی نزولی:

تجزیه بالا به پایین را می توان تلاشی در نظر گرفت که می خواهد برای یک رشته ورودی سمت چپ ترین اشتقاق را پیدا کند. عملاً یک تجزیه کننده بازگشتی نزولی به نام تجزیه کننده پیشگو را بررسی کردیم که نیاز به بازگشت به عقب نداشت. در اینجا ممکن است بازگشت به عقب نیز وجود داشته باشد. در حالت کلی یک parser بالا به پایین باید بتواند در صورت لزوم عمل backtracking (عقبگرد) را انجام دهد. به عنوان مثال در گرامر زیر:

$S \rightarrow cAd$

$A \rightarrow ab | a$

برای تولید cad = w با توجه به گرامر روبرو به صورت زیر عمل می شود:



backtarcking

- در روش های parser بالا به پایین عملگر تحلیل نحوی عملاً به صورت آزمایش و خطا انجام می‌گیرد parser های بالا به پایین معمولاً به صورت حریصانه عمل می‌کند یعنی با دریافت هر Token درخت تجزیه را تا حد ممکن گسترش می‌دهد و تنها زمانی که دیگر امکان گسترش درخت تجزیه وجود نداشته باشد Token بعدی را دریافت می‌کند. در parser های بالا به پایین درخت تجزیه از ریشه شروع شده و مراحل زیر به طور مکرر انجام می‌شود:

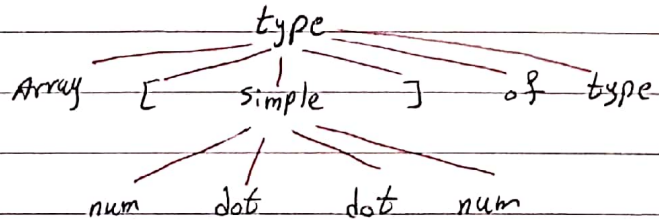
۱- در گره‌ی N برای غیرپایانی A یکی از قواعد A انتخاب شده و سمت راست این قاعده به عنوان فرزندان N در درخت پارس قرار می‌گیرد.

۲- گره‌ی بعدی را که از آنجا عمل تجزیه ادامه می‌یابد انتخاب کنیم به عنوان مثال گرامر زیر برای تعریف نوع یک کد در زبان pascal استفاده می‌شود در نظر بگیرید.

type → Simple | id | array [simple] of type

Simple → integer | chor | num dot dot num

در این مثال برای تولید جمله Array [num dot dot num] of type به صورت زیر عمل می‌کنیم:



subject : date :

به Token که parser در حال بررسی آن است Token ahead گفته می‌شود معمولاً انتخاب یک قاعده‌ی مناسب به روش صحیح و خطا انجام می‌شود و در صورت لزوم عمل عقبگرد یا backtracking انجام می‌گیرد به parser هایی که عمل عقبگرد را انجام نمی‌دهند و قاعده‌ی مناسب را به روش حدسی خاصی پیدا می‌کنند parser پیشگو گفته می‌شود.

روش پارسر پیشگو یک روش برای تولید درخت تجزیه است در این پارسر که به صورت بالا به پایین عمل می‌کند یک مجموعه از رویه‌ها به صورت بازگشتی رشته ورودی را مورد پردازش قرار می‌دهند این رویه‌ها یک درخت پارس ایجاد می‌کند. یک پارسر پایین گرد هر یک رویه دارد که دو کار انجام می‌دهد:

۱- تصمیم می‌گیرد که از کدام قاعده‌ی گرامر استفاده نماید

۲- از قاعده‌ی انتخاب شده استفاده می‌کند علاوه بر اینکه به ازای هر یک غیرپایانه یک رویه وجود دارد parser پایین گرد از رویه‌ی دیگری به نام match برای تطبیق Token های ورودی در حال ساخت استفاده می‌کند به عنوان مثال برای گرامری که در مثال قبل گفته شد دو رویه برای غیرپایانه‌های type و simple در نظر گرفته می‌شود همچنین رویه‌ی match که عمل تطبیق را انجام می‌دهد باید به این نکته اشاره گردد که با نوشتن توقیق یک گرامر خدف بازگشتی چپ و فاکتورگیری چپ از آن گرامر می‌توانیم گرامری بدست آوریم که توسط یک تجزیه کننده‌ی پیشگو قابل تجزیه است و احتیاجی به برگشت به عقب ندارد.

subject: date:

مثال: می خواهیم برای گرامر زیر یک الگوریتم نزولی تهیه کنیم. ابتدا یک تجزیه کننده پیشگو غیبا زگشتی بنویسیم، سپس کد تجزیه یک الگوریتم بازگشتی برای این گرامر است.

type → simple | id | array [simple] of type

simple → integer | char | num dot dot num

```
procedure Match (token t)
begin
    if lookahead = t then
        lookahead = next token
    else
        error;
end
procedure type
begin
  if lookahead is in <integer, char, num> then
   simple
else if lookahead is array then
begin
    match (array);
    match( [ );
    simple;
    match( ] );
    match (of);
    type;
end
else if lookahead is id then
      match (id);
else
    error;
end
```

subject: date:


```
procedure simple
begin
    if lookahead is integer then
        Match(integer)
else if lookahead is char then
        match(char);
else if lookahead is num then
begin
   match(num);
   Match (dot dot);
   Match (num);
    end
end
```

* پارسر پایین گرد:

برای پیشگویی قاعده‌ای که باید انتخاب کند از تابعی به نام first به روی سمت راست قواعد استفاده می‌کند. تابع می‌گوید برای هر پایانه و غیرپایانه‌ای محاسبه شود حاصل تابع first مجموعه‌ای از پایانه‌هایی است که در سمت چپ‌ترین بخش رشته‌های تولید شده توسط آن قرار می‌گیرد.

مثال: برای گرامر زیر first(S) و first(A) ، first(ab) را حساب کنید.

S → cAd

A → ab | b

first(S) = {c}

first(A) = {a, b}

first(ab) = {a}

مثال: برای گرامر زیر توابع First(simple) و First(id) و First(array[simple]of(type)) را بدست آورید.

type → simple | id | array [simple] of type

simple → integer | char | num dot dot num

First(simple) = {integer, char, num}

First(id) = {id}

First(array[simple] of type) = {array}

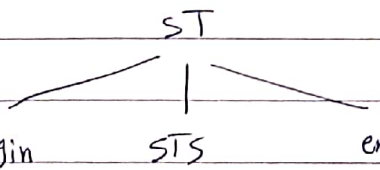
در گرامری که دارای دو قاعده به صورت $A \to \alpha$ و $B \to \beta$ می‌باشد parser باید بتواند با استفاده از first قاعده مناسب را انتخاب کند البته به شرطی که اشتراک firstα با firstβ برابر با $\emptyset$ (تهی) است

* با استفاده از عضو λ:

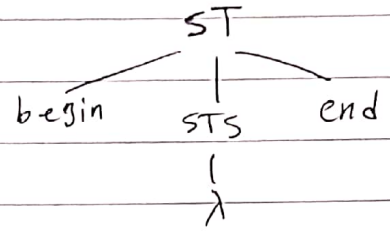
اگر در گرامری قاعده‌ای به صورت $A \to \lambda$ وجود داشته باشد parser باید بتواند از آن به عنوان قاعده‌ی پیش فرض استفاده کند یعنی اگر هیچ قاعده‌ی دیگری مناسب تشخیص داده نشود از قاعده‌ی λ استفاده می‌کنیم به عنوان مثال گرامر زیر را در نظر بگیرید اگر رشته‌ی ورودی begin end باشد درخت تجزیه بعد از دریافت Token begin و بسط ST به صورت زیر خواهد بود:

ST → begin STS end

STS → STL | λ

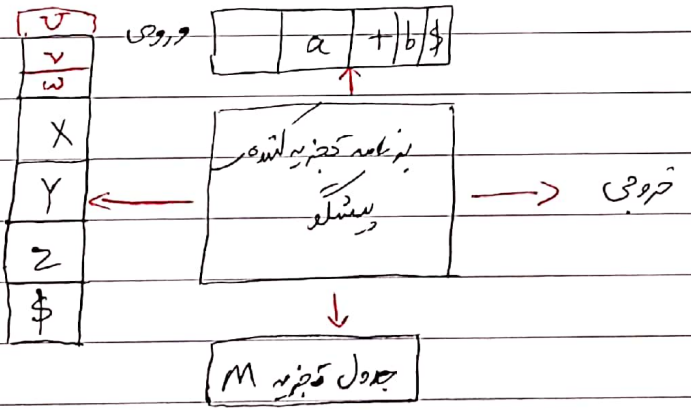


پس از بررسی Token ، begin بعدی یعنی end دریافت می‌شود پس در درخت تجزیه باید sts بسط یابد و چون end در first نیست بنابراین از قاعده λ برای بسط آن استفاده می‌شود دو نمایت رشته begin end تشخیص داده می‌شود



* تجزیه کننده پیشگو غیر بازگشتی:

ساختن یک تجزیه کننده غیر بازگشتی با ایجاد یک پشته و از طریق فراخوانی های بازگشتی امکان پذیر است مسئله کلیدی در تجزیه پیشگو تا این قاعده ی تولید استفاده شده برای یک غیر پایانی است تجزیه کننده غیر بازگشتی که در شکل زیر نشان داده می‌شود دنبال قانون تولیدی در جدول تجزیه می‌گردد تا از آن استفاده کند



تجزیه کننده پیشگویی که به کمک جدول تجزیه عمل می‌کند دارای بافر ورودی یک پشته یک جدول تجزیه و جریان خروجی است بافر ورودی شامل رشته ای است که باید تجزیه شود و به دنبال آن $ به عنوان علامت انتهایی نمبر است مورد استفاده قرار می‌گیرد تا پایان رشته ورودی را نشان دهد پشته شامل دنباله ای از نمادهای گرامر به همراه $ در پایین آن است که پایان پشته را مشخص می‌کند در آغاز پشته شامل نماد شروع گرامر در بالای $ می‌باشد

جدول تجزیه آرایه‌ای ۲ بعدی $M[A, a]$ است که در آن A بزرگ غیرپایانی و a پایانی یا $\$$ است، تجزیه‌کننده به وسیله برنامه‌ای کنترل می‌شود که به صورت زیر عمل می‌کند این برنامه X را به عنوان نماد بالای پشته و a را به عنوان نماد ورودی فعلی در نظر می‌گیرد این دو نماد عمل تجزیه را تعیین می‌کنند ۳ حالت ممکن زیر اتفاق می‌افتد:

۱- اگر $X=a=\$$ باشد تجزیه‌کننده متوقف می‌شود و موفقیت کامل تجزیه را اعلام می‌کند

۲- اگر $X=a\neq\$$ باشد تجزیه‌کننده X را از پشته خارج می‌کند و اشاره‌گر ورودی را نماد ورودی بعدی جلو می‌برد

۳- اگر X غیرپایانی باشد برنامه خانه $M[X,a]$ را از جدول تجزیه M بررسی می‌کند این خانه یا قانون تولید X از گرامر می‌باشد و یا خطا است.

برای مثال اگر $M[X,a]=\{X\rightarrow uvw\}$ تجزیه‌کننده M به جای X در بالای پشته uvw را که u در بالا قرار دارد می‌گذارد.

به عنوان خروجی فرض می‌کنیم تجزیه‌کننده فقط قوانین تولید مورد استفاده را چاپ می‌کند در این مرحله هر دستور دیگری می‌تواند جدا شود اگر $M[X,a]=Error$ باشد تجزیه‌کننده یک تابع تصحیح خطا را احضار می‌کند

* الگوریتم تجزیه پیشگوی غیربازگشتی:

ورودی رشته w و جدول تجزیه M برای گرامر G خروجی می‌دهد اگر w در $L(G)$ باشد ست چپ‌ترین اشتقاق w و در غیر این صورت خطا رخ می‌دهد

روش کار:

در آغاز تجزیه‌کننده ساختار دارد که در آن به همراه S ، $ در بالای پشته و نماد شروع G در بالا قرار دارد و $w در بافر ورودی است. این برنامه که از جدول تجزیه پیشگوی M برای تولید یک تجزیه برای ورودی استفاده می‌کند در ادامه آمده است:

```
set ip to point to the first symbol of w$;
repeat
    let X be the stop stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
    if X = a then
            pop X from the stack and advance ip;
    else error ()
else /* X is a nonterminal */
    if M[X, a] = X → Y1 Y2 ... Yk then begin
    pop X from the stack
    push Yk Yk-1, ..., Y1 onto the stack, with Y1 on top;
    output the production X → Y1 Y2 ... Yk
    end
    else error ()
until X = $ /* stack is empty */
```

subject : date :

مثال ۵ در گرامر زیر می‌خواهیم با قدرت id+id*id و جدول تجزیی پیشگو و بالا حرکت‌های ممکن را برای پذیرش این رشته ورودی بنویسید.

$E \rightarrow TE'$

$E' \rightarrow TE' \mid \lambda$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \lambda$

$F \rightarrow (E) \mid id$

input Symbol

Nonterminal	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'->+TE'			E'->λ	E'->λ
T	T->FT'			T->FT'		
T'		T'->λ	T'->*FT'		T'->λ	T'->λ
F	F->id			F->(E)		

stack	input	output
$E	id+id*id$	
$E'T	id+id*id$	E -> TE'
$E'T'F	id+id*id$	T -> FT'
$E'T'id	id+id*id$	F -> id
$E'T'	+id*id$	
$E'	+id*id$	T' -> λ
$E'T+	+id*id$	E' -> +TE'
$E'T	id*id$	
$E'T'F	id*id$	T -> FT'
$E'T'id	id*id$	F -> id
$E'T'	*id$	
$E'T'F*	*id$	T' -> *FT'
$E'T'F	id$	
$E'T'id	id$	F -> id
$E'T'	$	
$E'	$	T' -> λ
$	$	E' -> λ

First / Follow

ساخت یک تجزیه‌گر پیشگو به وسیله دو تابع مرتبط با گرامر G صورت می‌گیرد. این توابع First و Follow هستند که امکان پدید کردن درایه‌های جدول تجزیه‌ی پیشگو گرامر G را در صورت امکان فراهم می‌کنیم. مجموع نشانه‌های حاصل از تابع Follow را می‌توان به صورت نشانه‌های همزمان کننده در جریان تصحیح خطای حالت اضطراری به کار برد. اگر α هر رشته‌ای از نمادهای گرامر باشد $First\alpha$ مجموعه‌ای از پایانی‌هاست که رشته‌های مشتق شده از α با آن‌ها شروع می‌شود اگر α بعد از یک تعداد مراحل بدهد λ آنگاه λ نیز در $First\alpha$ است. $\alpha \overset{*}{\Rightarrow} \lambda$

$Follow A$ را برای غیرپایانی A به صورت مجموعه‌ای از پایانی‌های a تعریف می‌کنیم که می‌تواند بلافاصله در سمت راست A در یک شکل جمله‌ای ظاهر شود یعنی مجموعه‌ی پایانی‌های a به طوری که یک اشتقاق به صورت $S \overset{*}{\Rightarrow} \alpha A a \beta$ برای یک α و β وجود داشته باشد. توجه دارید که در برخی از موارد در جریان اشتقاق ممکن است نمادهایی بین A و a وجود داشته باشد اگر چنین باشد λ از آن‌ها مشتق شده و ناپدید می‌شود. اگر A بتواند سمت راست‌ترین نماد در یک شکل جمله‌ای باشد آنگاه $\$$ در $Follow A$ است به منظور محاسبه‌ی $First X$ برای تمامی نمادهای گرامر X قوانین زیر را تا آن جا مورد استفاده قرار می‌دهیم که امکان افزودن پایانی دیگر یا λ در هر مجموع $First$ وجود دارد:

۱) اگر X پایانی باشد آنگاه $First(X)$ برابر $\{X\}$ است.

۲) اگر $\lambda \leftarrow X$ یک قانون تولید باشد آنگاه λ را به $First X$ اضافه می‌کنیم.

۳) اگر X غیرپایانی باشد و $X \leftarrow Y_1 Y_2 \dots Y_k$ یک قانون تولید باشد آنگاه a را در صورتی در $First X$ قرار می‌دهیم که برای یک i a در $First(Y_i)$ باشد و λ در تمامی $First(Y_1) \dots First(Y_{i-1})$ قرار داشته باشد

۳) یعنی: $Y_1 \dots Y_{i-1} \overset{*}{\Rightarrow} \lambda$

اگر λ برای تمام j ها $1, \dots, i-1$ در $First(Y_j)$ باشد آنگاه λ را به $First(X)$ اضافه می‌کنیم حال می‌توان برای

هر رشته‌ی $X_1 X_2 \dots X_n$ ، $First$ را به صورت زیر محاسبه کنیم:

- تمام نمادهای مختلف λ از $First\,X_1$ به $First\,X_1 \dots X_n$ اضافه می‌کنیم، علاوه براین اگر λ در $First(X_1)$ باشد نمادهای

مخالف λ از $First(X_2)$ را اضافه کنید و اگر λ هم در $First(X_1)$ و هم در $First(X_2)$ باشد نمادهای مخالف λ را از $First(X_3)$

به آن اضافه کنید در پایان اگر برای تمام i ها $First(X_i)$ حاوی λ باشد، λ را به $First\,X_1 X_2 \dots X_n$ اضافه کنید.

- $Follow$:

به منظور محاسبه‌ی $Follow(A)$ برای تمام غیرپایانی‌های A قوانین را تا آنجایی استفاده می‌کنیم که امکان اضافه کردن چیزی در مجموع $Follow$ وجود داشته باشد:

۱) $\$$ در $Follow(S)$ قرار دارد که در آن S نماد شروع و $\$$ نماد پایان سمت راست ورودی است

۲) اگر قانون $A \to \alpha B$ یا قانون تولید $A \to \alpha B \beta$ وجود داشته باشد که در آن $First(\beta)$ حاوی λ است ($\beta \overset{*}{\Rightarrow} \lambda$)

آنگاه هر چیزی از $Follow\,A$ در $Follow\,B$ نیز است

مثال*: برای گرامر زیر مجموعه‌های $First/Follow$ را بدست آورید.

$E \to TE'$	$First(E) = First(T) = First(F) = \{(, id\}$
$E' \to +TE' \mid \lambda$	$First(E') = \{+, \lambda\}$
$T \to FT'$	$First(T') = \{*, \lambda\}$
$T' \to *FT' \mid \lambda$	$Follow(E) = Follow(E') = \{\$,)\}$
$F \to (E) \mid id$	$Follow(T) = Follow(T') = \{\$,), +\}$
	$Follow(F) = \{\$,), +, *\}$

* الگوریتم ساختن جدول تجزیه پیشگو:

ورودی: گرامر G

خروجی: جدول تجزیه M

* روش کار:

۱) برای هر قانون تولید $A \to \alpha$ از گرامر G مراحل ۲ و ۳ را انجام دهید.

۲) برای هر پایانی a در $First\alpha$ قانون $A \to \alpha$ را به $M[A,a]$ اضافه کنید

۳) اگر λ در $First\alpha$ است برای هر پایانی b در $FirstA$ ، $A \to \alpha$ به $M[A,b]$ اضافه کنید، اگر λ در $First\alpha$ و $\$$ در $FirstA$ باشد $A \to \alpha$ را به $M[A,\$]$ اضافه کنید

۴) برای هر درایه‌ی تعریف نشده M آن را برابر Error قرار دهید به عنوان مثال برای گرامر مثال قبل جدول تجزیه پیشگو به صورت زیر است:

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \lambda$	$E' \to \lambda$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \lambda$	$T' \to *FT'$		$T' \to \lambda$	$T' \to \lambda$
F	$F \to id$			$F \to (E)$		

* گرامرهای LL(1):

گرامر فوق را می‌توان برای هر گرامر G اعمال کرد تا جدول تجزیه M بدست آید با وجود این برای برخی از گرامرها خانه‌ای از جداول ممکن است تعاریف چندگانه داشته باشد که به صورت بازگشتی چپ یا ابهام در گرامر اصلی بوجود می‌آید:

$S \to iESS'|a$ $\quad First(S) = \{i, a\}$

$S' \to eS|\lambda$ $\quad First(S') = \{e, \lambda\}$

$E \to b$ $\quad First(E) = \{b\}$

$Follow(S) = \{\$, e\} = Follow(S')$

$Follow(E) = \{i, a\}$

* جدول نمای آخر فصل قبل:

	i	a	e	b	$
S	$S \to iEtSS'$	$S \to a$			
S'			$S' \to eS$ / $S' \to \lambda$		$S' \to \lambda$
E				$E \to b$	

$M[S', e] = ?$

ابهام دارد دوگرامر در یک خانه است و کامپایلر نمی‌تواند تشخیص دهد از کدام استفاده کند برای همین این گرامر LL(1) نیست

- گرامری که جدول تجزیه‌اش هیچ درایه‌ای با تعریف چندگانه ندارد را LL(1) می‌نامیم. L اول به معنی پردازش ورودی از چپ (Left) به راست و L دوم به معنی کوتاه‌ترین چپ‌ترین (Leftmost) اشتقاق و 1 به معنی استفاده از یک نماد ورودی از پیش‌نگاه در هر مرحله به منظور تصمیم‌گیری در عمل تجزیه است.

- الگوریتم ساختن یک جدول پیش‌گو برای هر گرامر LL(1) مانند گرامر G یک جدول تجزیه کوتاه می‌کند که تمام جملات G تجزیه می‌کند. گرامرهای LL(1) دارای چند ویژگی مشخص هستند. گرامرهای مبهم یا بازگشتی چپ نمی‌توانند از نوع LL(1) باشند. علاوه بر این می‌توان نشان داد که گرامر G، گرامر LL(1) است اگر و فقط اگر $A \to \alpha | \beta$ دو قانون تولید متمایز از G باشند و آنگاه شرایط زیر برقرار باشد:

1) برای هیچ پایانه a هم α و هم β رشته‌هایی را که با a شروع می‌شود تولید نکنند یعنی $First(\alpha) \cap First(\beta) = \phi$
2) حداکثر یکی از α و β بتوانند λ را تولید کنند
3) اگر β بتواند λ را تولید کند ($\beta \stackrel{*}{\Rightarrow} \lambda$)، آنگاه α رشته‌ای را تولید نکند که با پایانه‌ای که در $Follow(A)$ قرار دارد شروع شود یعنی:

$Follow(A) \cap First(\beta) = \phi$

مثال: آیا گرامر زیر LL(1) است؟

$E \to TE'$ — $First(E) = First(T) = First(F) = \{(, id\}$
$E' \to +TE' | \lambda$ — $First(E') = \{+, \lambda\}$
$T \to FT'$ — $First(T') = \{*, \lambda\}$
$T' \to *FT' | \lambda$ — $Follow(E) = Follow(E') = \{\$,)\}$
$F \to (E) | id$ — $Follow(T') = Follow(T) = \{\$,), +\}$
$Follow(F) = \{\$,), +, *\}$

$First(E') \Rightarrow \{First(+TE') = \{+\}, First(\lambda) = \{\lambda\}\}$ ← اشتراک این دو تا $= \phi$

$Follow(E') = \{\$,)\}$
$First(+TE') = \{+\}$ $\Rightarrow \cap = \phi$

$First(T') \Rightarrow \{First(*FT') = \{*\}, First(\lambda) = \{\lambda\}\} \cap = \phi$

$Follow(T') = \{+, \$,)\}$
$First(*FT') = \{*\}$ $\cap = \phi$

$First((E)) = \{(\}$
$First(id) = \{id\}$ $\cap = \phi$

$Follow(F) = \{*, +, \$,)\}$
$First((E)) = \{(\}$ $\cap = \phi$

- چون همه گرامرها First و Follow اشتراک آن‌ها ϕ پس LL1 است

* تصحیح خطا در تجزیه ی پیشگو:

روش اصلاح خطاهای نحوی در گرامر (LL(1 یکی از روش های متداول در اصلاح خطاهای نحوی روش panic mode (حالت اضطراری)، است. قبلاً گفتیم که در جریان تجزیه ی پیشگو خطا هنگامی تشخیص داده می شود که پایانی بالای پشته با نماد ورودی بعدی مطابقت نداشته باشد و یا غیرپایانی A در بالای پشته باشد a نماد ورودی بعدی است و درایه ی جدول تجزیه ی [M[A,a خالی باشد. تصحیح خطا در حالت panic mode بر این ایده استوار است که نمادهای ورودی را تا آن زمان کنار می گذاریم که یک نشانه در مجموعه ی انتخاب شده از نشانه های هماهنگ کننده (Synchronizing) ظاهر شود. کارایی این روش به انتخاب مناسب مجموعه ی هماهنگ کننده بستگی دارد. این مجموعه ها باید به گونه ای انتخاب شوند که تجزیه کننده خطاهایی را که احتمال وقوع آن ها وجود دارد به سرعت تصحیح کند. برخی از راه حل های آن به شرح زیر است:

۱) به عنوان نقطه ی شروع می توانیم تمام نمادهای Follow A را در مجموعه ی هماهنگ کننده ی A قرار دهیم. اگر نشانه ها تا زمانی دیدن یک عضو از Follow A کنار گذاشته شوند و A را از پشته خارج کنیم انتظار داریم تجزیه بتواند ادامه یابد.

۲) استفاده از Follow A به عنوان مجموعه ی هماهنگ کننده ی A کافی نیست. برای مثال اگر ; مانند زبان C پایان دستورات باشد آنگاه کلمات کلیدی که دستورات با آن شروع می شوند ممکن است در مجموعه ی Follow از غیرپایانی های تولید کننده ی عبارت ها ظاهر نشوند. فقدان ; (سمیکالن)، بعد از یک عمل جایگزینی ممکن است باعث عبور از کلمه ی کلیدی شروع کننده ی دستور بعدی شود.

۳) اگر نمادهای First A به مجموعه ی هماهنگ کننده ی مربوط به غیرپایانی A اضافه کنیم آنگاه از سرگرفتن تجزیه براساس A در صورتی امکان پذیر است که یک نماد از First A در ورودی ظاهر شود.

۴) اگر غیرپایانی بتواند رشته ی تهی را تولید کند آنگاه قانون تولیدی که λ از آن بدست می آید را می توان به صورت پیش فرض مورد استفاده قرار داد. انجام این کار ممکن است تشخیص خطا را معلق سازد اما نمی تواند باعث از دست رفتن خطا شود. این روش از تعداد غیرپایانی هایی که باید در جریان تصحیح خطا بررسی شوند می کاهد.

۵) اگر پایانه ی بالای پشته با Token جاری تطابق نداشته باشد پایانه ی بالای پشته حذف می شود.

واتسابی

مثال ← اگر رشته ورودی $id*+id)$ باشد آنگاه به دلیل اینکه رشته ورودی نادرست است خنثی کننده و مکانیزم تصحیح خطا به صورت زیر خواهد بود.

$E \to TE'$ — $First(E) = First(T) = First(F) = \{(, id\}$

$E' \to +TE' \mid \lambda$ — $First(E') = \{+, \lambda\}$

$T \to FT'$ — $First(T') = \{*, \lambda\}$

$T' \to *FT' \mid \lambda$ — $Follow(E) = Follow(E') = \{), \$\}$

$F \to (E) \mid id$ — $Follow(F) = \{+, *,), \$\}$

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch ← * در واقع Follow طباس
E'		$E' \to +TE'$			$E' \to \lambda$	$E' \to \lambda$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \lambda$	$T' \to *FT'$		$T' \to \lambda$	$T' \to \lambda$
F	$F \to id$	synch	synch	$F \to (E)$	synch	synch

Stack	Input	Remark
$\$E$	$)id*+id\$$	Error, skip)
$\$E$	$id*+id\$$	id is in First(E)
$\$E'T$	$id*+id\$$	
$\$E'T'F$	$id*+id\$$	
$\$E'T'id$	~~id~~$*+id\$$	
$\$E'T'$	$*+id\$$	
$\$E'T'F*$	~~$*$~~$+id\$$	
$\$E'T'F$	$+id\$$	Error, M[F, +] = synch
$\$E'T'$~~$F$~~	$+id\$$	F has been poped
$\$E'$~~$T'$~~	$+id\$$	
$\$E'T*$	$*id\$$	
$\$EE'T$	$id\$$	
$\$E'T'F$	$id\$$	

	$	$
	$	$
	$	$
	$	$

* تصحیح عبارت سطح (phase-level) :

آخرین روش اصلاح‌های خطاهای نحوی با (LL(1 که بالا به پایین بود روش phase-level است، این روش با پر کردن درایه‌های خالی در جدول تجزیه پیشگو با کمک اشاره‌گرهایی به روال‌های اصلاح خطا پیاده‌سازی می‌شود، این روال‌ها می‌توانند نماد ورودی را تغییر دهد درج یا اضافه کند و پیغام‌های خطاهای مناسبی بدهد. علاوه بر این ممکن است از پشته خارج شود.

- تجزیه‌ی پایین به بالا :

در این روش از یک روش کلی تحلیل ساختار دستور به نام تجزیه‌ی انتقال-کاهش (shift-reduce) استفاده می‌شود در این روش یا درخت‌های پایین به بالا از پشته‌ی ورودی شروع کرده و ساخت درخت تجزیه را از برگ‌ها به سمت ریشه (غیرپایانی شروع‌کرامر) ادامه می‌دهد می‌توان این فرایند را به عنوان کاهش رشته‌ی w به نماد شروع یک گرامر در نظر گرفت در هر مرحله از کاهش به جای یک زیررشته‌ی خاص که با سمت راست یک قانون تولید تطابق دارد نماد سمت چپ آن قانون تولید را قرار می‌دهیم و اگر در هر مرحله زیررشته به درستی انتخاب شود سمت راست‌ترین اشتقاق به صورت عکس ردیابی می‌شود.

نکته* ترتیب بکارگیری قواعد در روش‌های بالا به پایین از چپ به راست است ولی در الگوریتم‌های پایین به بالا از راست به چپ می‌باشد.

مثال: گرامر زیر را در نظر بگیرید. سپس راست گرام برای تولید رشته abbcde به صورت زیر است

S -> aABe

A -> Abc | b

B -> d

اشتقاق چپ: $S \underset{L-m}{\Rightarrow} aABe \Rightarrow aAbcBe \Rightarrow abbcBe \Rightarrow abbcde$

Left-most

اشتقاق راست: $S \underset{R-m}{\Rightarrow} aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

Right-most

روش بالا به پایین:

abbcde

aAbcde

aAde

aABe

S

* عبارت pharse:

بخشی از یک فرم جمله‌ای است که از یک متغیر با چی تولید می‌شود:

$S \overset{*}{\Rightarrow} \alpha A \gamma \overset{+}{\Rightarrow} \alpha\beta\gamma$

* عبارت ساده (Simple pharse):

عبارتی است که در یک فرم تولیدی می‌شود:

$S \overset{*}{\Rightarrow} \alpha A \gamma \Rightarrow \alpha\beta\gamma$

* دسته (Handel):

عبارت ساده‌ای است که در جهت عکس یک راست تولید می‌شود:

$S \overset{*}{\Rightarrow} \alpha A_2 \Rightarrow \alpha\beta_2$

گر یک دستگیره است و نه یک پارس، اگر گرامر کتگ نباشد، در هر مرحله فقط یک دستگیره تولید می‌شود. در گرامرهای کتگ در برخی موارد بیش از یک دستگیره داریم.

مثال: گرامر زیر را در نظر بگیرید.

این گرامر کتگ است زیرا برای تولید id+id*id دو بسط راست و در نتیجه دو مسیر برای پارسر پایین به بالا وجود دارد

$E \to E+E$

$E \to E*E$

$E \to (E)$

$E \to id$

بسط راست $\Rightarrow E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*id \Rightarrow E+id*id \Rightarrow id+id*id$

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

نتیجه آنکه مبهم است و پارسر پایین به بالا نمی‌تواند با آن کار کند

* در روش انتقال-کاهش از یک stack و یک بافر برای ذخیره‌ی رشته ورودی استفاده می‌شود. در ابتدای پارس، در انتهای رشته ورودی و در بالای stack علامت $ می‌گذاریم. عملیات انتقال-کاهش تا زمانی انجام می‌شود که یا پارسر با موفقیت به پایان برسد یا یک خطای نحوی رخ دهد. در انتهای پارس در رشته ورودی $ باقی می‌ماند و در stack نیز علامت شروع گرامر در کنار $ قرار گرفته است و به طور کلی عملیات زیر انجام می‌گیرد:

- عمل کاهش:

دسته‌گیره‌ی بالای stack حذف می‌شود و به جای آن غیرپایانی سمت چپ قاعده‌ای که سمت راست آن متناظر با دسته‌گیره است قرار می‌گیرد

- عمل خطا:

یک خطای نحوی در پارسر رخ داده است

- عمل انتقال:

بخشی از رشته ورودی به stack منتقل می‌شود. انتقال زمانی پایان می‌یابد که یک دسته‌گیره در stack تشخیص داده شود.

- عمل پذیرش:

پارسر پایان موفقیت آمیز پارس را اعلام می کند

مثال: رشته‌ی id+id را به روش انتقال کاهش تجزیه کنید.

E -> E+E

E -> E*E

E -> (E)

E -> id

stack	input	Remark
$	id+id $	محل انتقال id
$id	+id $	کاهش E -> id
$ E	+id $	انتقال +
$ E+	id $	انتقال id
$ E+id	$	کاهش E -> id
$ E*E	$	کاهش E -> E+E
$ E	$	
Accept		

نکته * در روش انتقال کاهش مشکلات زیر وجود دارد:

۱) تصمیم گیری درباره‌ی این که کدام زیر رشته تشکیل یک دستگیره را می دهد

۲) تایین این که از کدام قاعده برای کاهش استفاده شود، بنابراین دو نوع تداخل ایجاد می شود:

الف- تداخل انتقال - کاهش:

زمانی ایجاد می شود که پارسر نتواند تشخیص دهد که عمل انتقال باید انجام شود یا نه

مثال *

stmt -> if expr then stmt

| if expr then stmt else stmt

| other

اگر Token جاری else و رشته‌ی if expr then stmt در بالای پشته باشد پارسر می تواند هم با توجه به قاعده‌ی اول عمل کاهش را انجام دهد و هم می تواند عمل انتقال else به پشته را انجام دهد و بعداً در موقعیت مناسب عمل کاهش را انجام دهد.

ب - تداخل کاهش - کاهش:

زمانی ایجاد می‌شود که بیشتر از یک قاعده برای کاهش داشته باشیم یعنی سمت راست قاعده، سمت راست بیش از یک قاعده مطابق با دستگیره باشد.