کامپایلر

مجموعهٔ مهندسی کامپیوتر

دکتر غلامرضا قاسم ثانی

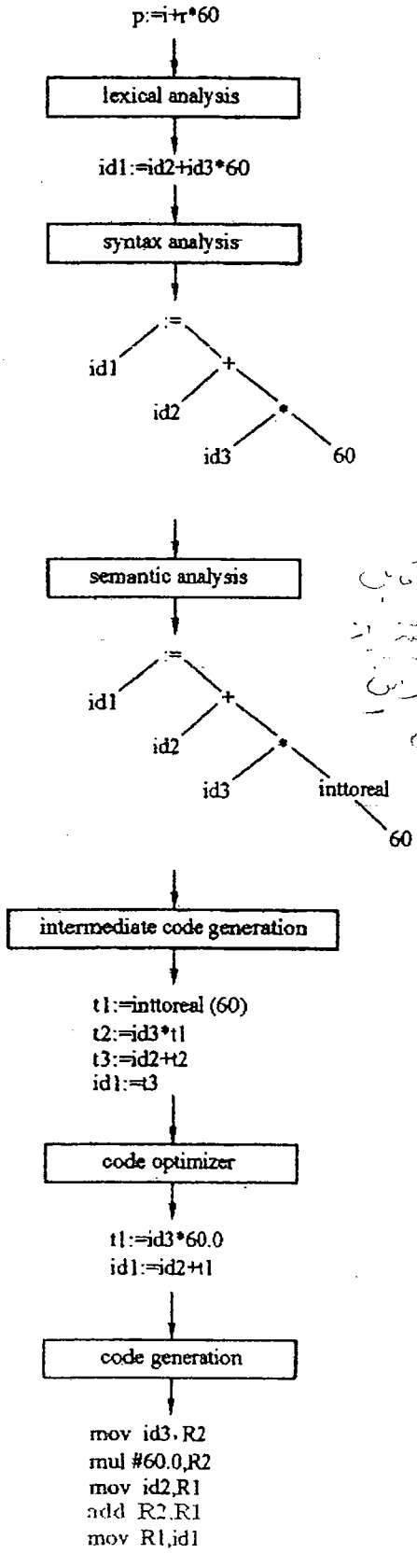
مؤسسهٔ آموزش عالی آزاد پارسه

ویرایش چهارم: بهار ۸۶ | تیراژ: ۲۰۰۰ نسخه |

شابک: ۰ - ۳۲ - ۸۷۱۹ - ۹۶۴ | ISBN: 964 - 8719 - 33 - 0

نشانی: بالاتر از میدان ولی‌عصر | کوچه دانش کیان | ساختمان پارسه | تلفن: ۸۸۸۴۹۲۱۱

اصول طراحی و ساخت کامپایلرها ۱
تعریف ۱
مراحل کامپایل ۱
خطا پرداز ۲
گرامرها ۵
تحلیل واژه‌ای ۸
خطاهای واژه‌ای ۹
روش‌هایی جهت بهبود کار اسکنر ۱۰
استفاده از میانگیر ۱۰
استفاده از نگهبان ۱۱
عملیات روی زبان‌ها ۱۲
عبارات منظم ۱۳
دیاگرام‌های انتقال ۱۵
تجزیه نحوی بالا به پایین ۲۲
تجزیه‌ی پایین‌گرد ۲۴
مشکل چپ‌گردی ۲۶
حذف چپ‌گردی ضمنی ۲۷
فاکتورگیری از چپ ۲۸
تجزیهٔ پیش‌گویانه غیربازگشتی LL (1) ۳۱
توابع First و Follow ۳۲


p:=i+r*60
lexical analysis
id1:=id2+id3*60
syntax analysis
:=
id1
+
id2
*
id3
60
semantic analysis
:=
id1
+
id2
*
id3
inttoreal
60
intermediate code generation
t1:=inttoreal (60)
t2:=id3*t1
t3:=id2+t2
id1:=t3
code optimizer
t1:=id3*60.0
id1:=id2+t1
code generation
mov id3,R2
mul #60.0,R2
mov id2,R1
add R2,R1
mov R1,id1

گرامرها

با توجه به اهمیت زیادی که مفهوم دستور زبان یا گرامر در ارتباط با عمل ترجمه دارد، در این بخش به معرفی اجمالی مفهوم دستور زبان می‌پردازیم.

تعریف

یک گرامر، مجموعه متناهی است از قواعدی که ساخت جملات یک زبان (Language) را مشخص می‌کند. در واقع گرامر به ما کمک می‌کند که تشخیص دهیم آیا جمله‌ای به یک زبان متعلق است یا خیر. پس هر گرامر معرف یک زبان است.

گرامر به‌وسیله چهارتایی $G=<N,T,S,P>$ تعریف می‌شود که در آن N مجموعه غیرپایانه‌ها (Nonterminal) و T مجموعه پایانه‌ها (Terminal) می‌باشند. S یک عضو ثابت و مشخص از N است که علامت شروع گرامر (Starting symbol) نامیده می‌شود و P یک مجموعه متناهی است که قواعد زبان را در بر می‌گیرد.

مثال : گرامر $G1=<N,T,S,P>$ را در نظر بگیرید:

$N = \{E\}$

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

$S = E$

در این مثال، E علامت شروع گرامر است. در صورتی که علامت شروع گرامری مشخص نباشد، اولین غیر پایانه‌ای که در گرامر ظاهر می‌شود، به عنوان علامت شروع در نظر گرفته می‌شود.

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

قرارداد: از این پس غیر پایانه‌ها را با حروف بزرگ الفبای لاتین و پایانه‌ها را با حروف کوچک نشان می‌دهیم.

رشته با طول صفر را نیز با ε نشان می‌دهیم.

به‌طور کلی، زبانی که گرامر G توصیف می‌کند، به صورت زیر تعریف می‌شود:

$$L(G) = \{\alpha \mid S \Rightarrow^{+} \alpha \text{ و } \alpha \in T^{*}\}$$

یعنی زبان L(G)، مجموعه‌ای شامل رشته‌هایی مثل α است که اگر از علامت شروع گرامر شروع کنیم، بر اساس قواعد گرامر G در یک قدم یا بیش‌تر تولید می‌شوند $(S \Rightarrow^{+} \alpha)$ و α رشته‌ای است به طول صفر یا بیش‌تر از پایانه‌ها $(\alpha \in T^{*})$

مجموعه فرم‌های جمله‌ای (Sentential Form) یک زبان را به صورت زیر تعریف می‌کنیم:

$$SF(G) = \{\beta \mid S \Rightarrow^{*} \beta, \beta \in V^{*}\}$$

به این معنی، که فرم جمله‌ای گرامر G شامل رشته‌هایی است مانند β به‌طوری که از علامت شروع گرامر شروع کنیم، بتوانیم در صفر قدم یا بیش‌تر رشته β را تولید کنیم $(S \Rightarrow^{*} \beta)$ و β رشته‌ای به طول صفر یا بیش‌تر از پایانه‌ها و غیر پایانه‌ها است $(V = N \cup T)$. با توجه به تعاریف فوق مشخص است که $L(G) \subset SF(G)$.

انواع گرامر:

چامسکی (Chomski) با در نظر گرفتن محدودیت‌هایی بر روی فرم قواعد، گرامرها را به چهار گروه زیر تقسیم کرد:

- گرامر نوع صفر یا بدون محدودیت (Unrestricted)
- گرامر نوع ۱ یا حساس به متن (Context Sensitive)
- گرامر نوع ۲ یا مستقل از متن (Context Free)
- گرامر نوع ۳ یا منظم (Regular)

گرامرهای نوع صفر، شامل قواعدی به فرم $\alpha \rightarrow \beta$ هستند و محدودیت خاصی ندارند، به غیر از آن‌که سمت چپ این قواعد باید حداقل شامل یک غیر پایانه باشد.

در گرامرهای حساس به متن، علاوه بر محدودیت فوق ، بایستی $|\alpha| \leq |\beta|$ باشد؛ یعنی طول (تعداد علایم) رشته سمت راست یک قاعده نبایستی کمتر از طول سمت چپ آن باشد.

در گرامرهای مستقل از متن، علاوه بر محدودیت‌های فوق، بایستی $|\alpha| = 1$ باشد. به عبارت دیگر، سمت چپ قواعد این نوع گرامرها فقط، یک غیرپایانه است.

بالاخره، در گرامرهای نوع ۳، علاوه بر محدودیت‌های فوق، β می‌تواند فقط به‌صورت (a و Ba) و یا (a و aB) باشد که در آن $a \in T$ و $B \in N$ است.

تعریف BNF – (Backus Normal From) قواعد یک گرامر را می‌توان با استفاده از فرم BNF که اولین‌بار در توصیف زبان برنامه‌سازی Algol 60 استفاده شد، نشان داد. در BNF غیر پایانه‌ها بین دو علامت < و > قرار داده می‌شوند و به‌جای علامت $\rightarrow$ نیز از علامت ::= استفاده می‌شود. سمت راست قواعدی را که سمت چپ یکسانی دارند نیز با علامت "|" به معنی "یا" جدا می‌شود.

<identifier > ::=<letter>|<identifier> <letter>|<identifier> <digit>
<letter> ::=a|b|...|y|z
<digit> ::=0|1|2|...|9

بسط (Derivation)

بسط، عملی است که در آن از علامت شروع گرامر آغاز کرده و با استفاده از قواعد گرامر یک فرم جمله‌ای تولید می‌گردد. دو روش بسط قانونمند (Canonical) وجود دارد:

بسط چپ (Left Most Derivation) که در هر قدم از آن، سمت چپ‌ترین غیر پایانه‌ای را که در فرم جمله‌ای است، با استفاده از قواعد آن غیرپایانه بازنویسی می‌کنیم.

بسط راست (Right Most Derivation) که در هر قدم از آن، سمت راست‌ترین غیر پایانه موجود در فرم جمله‌ای را با کمک یکی از قواعد آن غیرپایانه بازنویسی می‌کنیم.
به عنوان نمونه گرامر زیر را در نظر بگیرید:

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

بسط چپ جمله id+id*id به صورت زیر است:

$$E \Rightarrow^{1} E+E \Rightarrow^{4} id+E \Rightarrow^{2} id+E*E \Rightarrow^{4} id+id*E \Rightarrow^{4} id+id*id$$

بسط راست برای جمله فوق به‌صورت زیر خواهد بود:

$$E \Rightarrow^{1} E+E \Rightarrow^{2} E+E*E \Rightarrow^{4} E+E*id \Rightarrow^{4} E+id*id \Rightarrow^{4} id+id*id$$

گرامرهای گنگ (Ambiguous)

در صورتی که با استفاده از قواعد یک گرامر بتوانیم برای یک جمله، دو بسط چپ مختلف و یا دو بسط راست متفاوت پیدا کنیم، آن گرامر گنگ است.

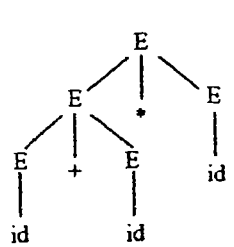
مثال :

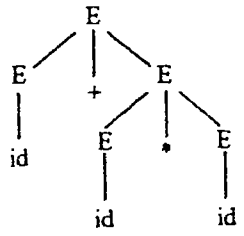
دو بسط چپ زیر برای جمله id + id * id وجود دارد.

$$E \Rightarrow^{1} E+E \Rightarrow^{4} id+E \Rightarrow^{2} id+E*E \Rightarrow^{4} id+id*E \Rightarrow^{4} id+id*id$$

$$E \Rightarrow^{2} E*E \Rightarrow^{1} E+E*E \Rightarrow^{4} id+E*E \Rightarrow^{4} id+id*E \Rightarrow^{4} id+id*id$$

هم‌چنین دو درخت پارس متفاوت نیز می‌توان برای این جمله رسم کرد.



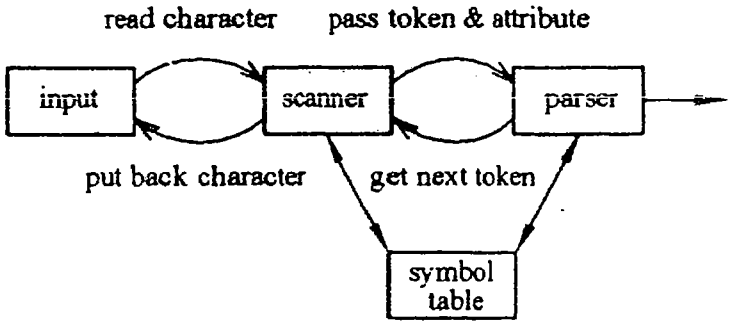


گرامر زیر، معادل گرامر فوق است، لیکن گنگ نیست. دو گرامر معادل محسوب می‌شوند، اگر چنان‌چه دقیقاً یک زبان را توصیف کنند.
بنابراین، گرامر زیر با 6 قاعده تولید، دقیقاً همان زبانی را توصیف می‌کند که گرامر فوق با 4 قاعده تولید، توصیف می‌کند.

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

تحلیل واژه‌ای (Lexical Analysis)

نخستین مرحله کامپایل تحلیل واژه‌ای است. به واحدی از کامپایلر که کار تحلیل واژه‌ای را انجام می‌دهد، اسکنر (Scanner) می‌گویند. اسکنر بین رشته ورودی و تحلیل‌گر نحوی یا پارسر (Parser) واقع است. وظیفه اصلی اسکنر، این است که با خواندن کارکترهای ورودی، توکن‌ها را تشخیص داده و برای پارسر ارسال نماید. رابطه اسکنر و پارسر به‌صورت زیر است:



به عنوان مثال، در صورتی که رشته ورودی A := B + C باشد، توکن‌های زیر تشخیص داده خواهد شد:

< آدرس C در جدول علایم . id > و <add op.> و < آدرس B در جدول علایم . id > و <, ass . op > و < آدرس A در جدول علایم .id>

بنابراین، اسکنر علاوه بر تشخیص این که توکن یک شناسه است، آدرس آن در جدول نشانه‌ها را نیز برای پارسر می‌فرستد. علاوه بر این، اسکنر می‌تواند محل‌های خالی و توضیحات (Comments) موجود در برنامه اصلی را ضمن خواندن برنامه حذف نماید.

به آخرین توکنی که اسکنر یافته است، علامت پیش‌بینی (Lookahead Symbol) و یا توکن جاری گفته می‌شود.

الگو (Pattern) و واژه (Lexeme) توکن‌ها

به فرم کلی که یک توکن می‌تواند داشته باشد، الگوی (Pattern) آن توکن می‌گویند. به عبارتی دیگر، در ورودی، رشته‌هایی وجود دارند که توکن یکسانی برای آن‌ها تشخیص داده می‌شود. فرم کلی این رشته‌ها توسط الگوی آن توکن توصیف می‌شود.

به دنباله‌ای از نویسه‌ها که تشکیل یک توکن می‌دهند، واژه (Lexeme) آن توکن می‌گویند. جدول زیر چند نمونه الگو و واژه است.

Token	Lexeme	Pattern
Const	Const	Const
Relation	< , > , <= , >= , < > , =	< or > or < = or > = or < > or =
Identifier	Pi, a25....	با حروف الفبا شروع و به‌دنبال آن صفر یا چند حرف و رقم قرار می‌گیرد
Num	3.1416, 12. ...	هر ثابت عددی
literal	"core dumped"	هر رشته نویسه‌ای که بین دو علامت " قرار گیرد.

در بعضی وضعیت‌ها، اسکنر قبل از این که تصمیم بگیرد چه توکنی را به پارسر بفرستد، نیاز دارد که چند کارکتر دیگر نیز از ورودی بخواند. به عنوان مثال اسکنر با دیدن علامت '<' در ورودی نیاز دارد که کارکتر ورودی بعدی را نیز بخواند. در صورتی که این کارکتر '=' باشد، توکن '=<' و در غیر این صورت توکن '<' تشخیص داده می‌شود. در مورد دوم، باید کارکتر اضافی خوانده شده دوباره به ورودی [illegible]

یکی دیگر از مشکلاتی که اسکنر با آن روبرو است، در زبان‌هایی نظیر Fortran پیش می‌آید. در این قبیل زبان‌ها محل خالی (blank) به‌جز در رشته‌های کارکتری نادیده گرفته می‌شود. به عنوان مثال، کلمه Do در فرترن در دستور زیر را در نظر بگیرید. Do5I=1.25 تا زمانی که نقطه اعشار در 1.25 دیده نشود، نمی‌توان گفت که Do در این دستور کلمه کلیدی نیست، بلکه بخشی از متغیر Do5I است. به همین ترتیب، در دستور Do5I=1,25 تا زمانی که علامت کاما دیده نشود، نمی‌توان گفت که این یک حلقه Do است.

در زبان‌هایی که در آن‌ها کلمات کلیدی (Keyword) جزو کلمات رزرو شده نیستند، نظیر زبان PL/I ، اسکنر نمی‌تواند کلمات کلیدی را از شناسه‌های همنام آن‌ها تشخیص دهد. به عنوان مثال در دستور زیر:

IF Then THEN Then = Else ELSE Else=Then;

جدا کردن کلمه کلیدی THEN از متغیر Then بسیار مشکل است. در این موارد، معمولاً پارسر تشخیص نهایی را خواهد داد.

خطاهای واژه‌ای (Lexical Errors)

به‌طور کلی، اسکنر، خطاهای محدودی را می‌تواند بیابد. زیرا اسکنر، تمام برنامه ورودی را یک‌جا نمی‌بیند؛ بلکه هر بار قسمتی کوچکی از برنامه منبع را به‌عنوان ورودی مورد بررسی قرار می‌دهد. به عنوان مثال، هر گاه رشته fi در یک برنامه C برای بار اول مشاهده شود، اسکنر قادر نیست تشخیص دهد که آیا fi یک املای غلط از کلمه کلیدی if است، یا یک متغیر:

fi(x= = f(x))

از آن‌جایی که fi یک متغیر مجاز است، اسکنر این توکن را به عنوان یک شناسه به پارسر می‌فرستد، تا این که پارسر در این مورد تصمیم بگیرد. اما ممکن است خطاهایی پیش بیاید که اسکنر قادر به انجام هیچ عملی نباشد. در این حالت، برنامه خطاپرداز (Error – handler) فراخوانده می‌شود تا آن خطا را به نحوی برطرف کند. روش‌های مختلفی برای این کار وجود دارد که ساده‌ترین آن‌ها، روشی موسوم به "panic mode" است. در این روش آن‌قدر از رشته ورودی حذف می‌شود تا اینکه یک توکن درست تشخیص داده شود.

سایر روش‌های تصحیح خطا (Error-Recovery) عبارتند از:

۱ـ حذف یک کارکتر غیرمجاز مثلاً (تبدیل = $: به =:)

۲ـ وارد کردن یک کارکتر گم شده به عنوان مثال (تبدیل : به =)

۳ـ تعویض کردن یک کارکتر غلط با یک کارکتر درست مثلاً (تبدیل :: به =:)

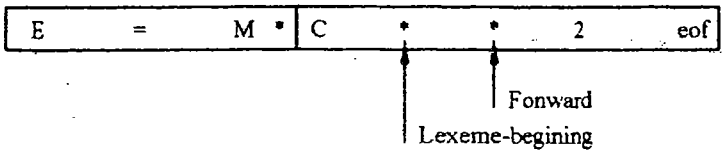
۴ـ جابه‌جا کردن دو کارکتر مجاز نظیر (تبدیل := به =)

روش‌هایی جهت بهبود کار اسکنر

استفاده از میان‌گیر

در بسیاری از زبان‌ها، اسکنر برای تشخیص نهایی توکن‌ها و مطابقت دادن آن با الگوهای موجود نیاز دارد که چند کاراکتر جلوتر را نیز مورد بررسی قرار دهد. از آن‌جایی که مقدار زیادی از زمان، در جابه‌جایی کارکترها سپری می‌شود، از تکنیک‌های استفاده از میان‌گیر (بافرینگ) برای پردازش کارکترهای ورودی استفاده می‌گردد.

در یکی از این تکنیک‌ها از یک میان‌گیر که به دو نیمه N کارکتری تقسیم شده است، استفاده می‌شود؛ N تعداد کارکترهایی است که در روی یک بلوک از دیسک جای می‌گیرند. به این ترتیب که با یک فرمان read، به جای خواندن یک کارکتر می‌توان N کارکتر ورودی را در هر نیمه بافر قرار داد. در صورتی که ورودی، شامل تعداد کاراکترهای کمتر از N باشد، بعد از خواندن این کارکترها، یک کارکتر خاص eof نیز وارد میان‌گیر می‌گردد. کارکتر eof بیان‌گر پایان فایل منبع بوده و با سایر کارکترهای ورودی تفاوت دارد.



در بافر ورودی از دو نشانه‌رو استفاده می‌شود. رشته کارکتری بین این دو نشانه‌رو، معرف Lexeme جاری می‌باشد. در ابتدا هر دو نشانه‌رو به اولین کارکتر Lexeme بعدی که باید پیدا شود، اشاره می‌کنند. نشانه‌روی Forward پیش می‌رود تا این که یک توکن تشخیص داده شود.

این نحوه استفاده از میان‌گیر در بیش‌تر موارد کاملاً خوب عمل می‌کند. با این وجود، در مواردی که جهت تشخیص یک توکن نشانه‌رو Forward ناچار است بیش‌تر از طول N کارکتر جلو برود، این روش کار نمی‌کند. به عنوان مثال، دستور DECLARE (ARG1 , ARGE2,....ARGn) را در یک برنامه PL/I در نظر بگیرید. در این دستور، تا زمانی که کارکتر بعد از پرانتز سمت راست را بررسی نکنیم، نمی‌توان گفت که DECLARE یک کلمه کلیدی است و یا یک اسم آرایه.

برای کنترل حرکت نشانه‌روی Forward و هم‌چنین کنترل بافر می‌توان به صورت زیر عمل کرد:

```
if  Forward is at end of first half Then begin
      reload second  half;
     Forward = Forward + 1
end
```

در این قسمت اگر نشانه‌رو Forward در انتها نیمه اول بافر رسید، نیمه دوم بافر با N کارکتر جدید پر خواهد شد.

```
else if Forward is at end of second half Then begin
      reload first half;
   move Forward to beginning of first half
end
else Forward := Forward+1;
```

در صورتی که نشانه‌رو Forward، به انتهای نیمه دوم بافر رسید، نیمه اول بافر را با N کارکتر جدید پر می‌کنیم و نشانه‌رو Forward را به آغاز بافر منتقل می‌کنیم.

استفاده از نگهبان‌ها (Sentinels)

در حالت قبل، با جلو رفتن نشانه‌رو Forward باید چک می‌شد که آیا این نشانه‌رو به انتهای یک نیمه از بافر رسیده است یا خیر. در صورتی که به انتهای یک نیمه بافر رسیده باشد، باید نیمه دیگر را دوباره پر می‌کردیم. در الگوریتم فوق، همان‌طوری که مشاهده شد، برای هر جلو روی نشانه‌رو Forward دو عمل مقایسه انجام می‌شود. می‌توان این دو را به یک بار تست تبدیل کرد. برای این کار باید در انتهای هر نیمه بافر یک کارکتر نگهبان (Sentinel) قرار دهیم.

E	•	M	•	eof	C	•	•	2	eof	eof

به این ترتیب، برای کنترل حرکت نشانه‌رو Forward می‌توانیم از الگوریتم زیر استفاده کنیم.

```
Forward : = Forward +1;
If ↑ Forward=eof Then begin
 if Forward is at end of First half Then begin
      reload second half :
       Forward := Forward +1
end
else if Forward is at  end of second half Then begin
       reload First half;
       move Forward to beginning of First half      Forward := 0
    end
    else
  /*eof within a buffer signifying end of input*/
   terminate lexical analysis
end
```

در این برنامه، ابتدا Forward یک خانه، جلو برده می‌شود. بعد محتوای خانه‌ای که Forward به آن اشاره می‌کند با کاراکتر eof مقایسه می‌شود. اگر در خانه مزبور، eof قرار داشته باشد، با انجام تست‌های دیگر مشخص خواهد شد که آیا به انتهای یکی از دو نیمه بافر رسیده‌ایم و یا این‌که به انتهای ورودی رسیده‌ایم. اما در اکثر موارد، همان تست اول نشان می‌دهد که خانه مورد اشاره حاوی eof نیست و در نتیجه تست‌های دیگر انجام نمی‌شود.

در ادامه یک سری تعاریف مرتبط با مفهوم زبان ارایه می‌گردد:

الفبا (Alphabet): یک مجموعه محدود از علایم را گویند. مثلاً:

$L = \{a..... z\}$

$D = \{0.....9\}$

رشته (String): یک دنباله محدود از علایم عضو یک الفبا را گویند. طول رشته S را با |S| نشان می‌دهند که برابر با تعداد علایم موجود در رشته است. یک رشته خود از بخش‌هایی تشکیل شده است. این بخش‌ها عبارتند از:

پیشوند (Prefix): رشته حاصل از حذف صفر یا چند علامت، از انتهای رشته S را گویند، مثلاً banna یک پیشوند از bannana است.

پسوند (Suffix): رشته حاصل از حذف صفر یا چند علامت، از ابتدای رشته S را گویند، مثلاً nana یک پسوند از bannana است.

زیر رشته (Substring): رشته حاصل از حذف یک پیشوند و یک پسوند از رشته S را گویند. مثلاً nan یک زیر رشته از bannana است. هر پیشوند و هر پسوندی خود نیز یک زیر رشته‌اند.

برای هر رشته‌ای مانند S، خود S، هر زیر رشته‌ای از S و هر پسوند یا پیشوند S، همگی زیر رشته S هستند.

زبان (Language): مجموعه‌ای از رشته‌ها را یک زبان گویند.

عملیات روی زبان‌ها

برخی از عملیاتی که روی زبان‌ها تعریف می‌شوند، عبارتند از عمل، اجتماع، الحاق و بستار که به‌صورت زیر تعریف می‌شوند:

عمل	تعریف
اجتماع union	$LUM = \{s \mid s \in L \text{ or } s \in M\}$
الحاق (Concatination)	$LM = \{st \mid s \in L \text{ and } t \in M\}$
بستار (Closure)	$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \ldots$
بستار مثبت (Positive Closure)	$L^+ = L^1 \cup L^2 \cup L^3 \ldots$

عمل توان برای زبان‌ها به‌صورت زیر تعریف می‌شود:

$L^0 = \{\varepsilon\}$

$L^i = L^{i-1}L$

برای مثال، فرض کنید:

$D = \{0,1,\ldots,9\}$

$L = \{A, B, \ldots Z, a, b, \ldots z\}$

در این صورت خواهیم داشت:

$LUD = \{A, B, \ldots Z, a, b, \ldots z, 0, 1, \ldots, 9\}$

$LD = \{A0, A1, \ldots, A9, B0, B1, \ldots B9, \ldots Z0, Z1, \ldots Z9, a0, a1, \ldots a9, \ldots z1, \ldots z9\}$

$L^4 = \{AAAA, ABCD, Abcd, \ldots\}$

$L = \{\varepsilon, A, AA, Bcd, \ldots\}$

$L(LUD)^* = \{A, AA1, A1, \ldots\}$

عبارات منظم (Regular Expression)

عبارات منظم روی الفبا Σ به صورت زیر تعریف می‌شود: (هر عبارت منظم r یک زبان L (r) را تعریف می‌کند.)

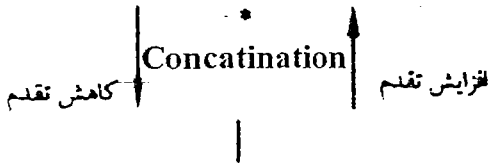
۱ـ ε یک عبارت منظم است که زبان $\{\varepsilon\}$ را تعریف می‌کند.

۲ـ اگر a عضو Σ باشد، a یک عبارت منظم است که زبان $\{a\}$ را تعریف می‌کند.

۳ـ اگر r و s عبارات منظمی باشند که زبان‌های L (r) , L(s) را تعریف می‌کنند، آن‌گاه:

- $(r) \mid (s)$ یک عبارت منظم است و زبان $L(r) \cup L(s)$ را تعریف می‌کند.
- $(r)(s)$ یک عبارت منظم است و زبان $L(r)L(s)$ را تعریف می‌کند.
- $(r)^*$ یک عبارت منظم است و زبان $L(r)^*$ را تعریف می‌کند.
- (r) یک عبارت منظم است و زبان L(r) را تعریف می‌کند. به این معنی که پرانتز در معنای عبارت منظم تأثیری ندارد و از آن برای مشخص کردن تقدم عملیات استفاده می‌شود.

زبان تعریف شده توسط یک عبارت منظم را " مجموعه منظم" یا " Regular Set " گویند. تقدم عملیات در عبارات منظم به‌صورت زیر است:



مثال : عبارت منظم $(a)\mid((b)^*(c))$ با عبارت $a \mid b^*c$ معادل است. به‌طور کلی، دو عبارت منظم را معادل گویند، اگر هر دو یک زبان را تعریف کنند. مثلا: $(a \mid b) = (b \mid a) = \{a, b\}$

در جدول زیر برخی از خصوصیات جبری عبارات منظم آورده شده است.

رابطه	توضیح
$r \mid s = s \mid r$	خاصیت جابه‌جایی|
$r \mid (s \mid r) = (r \mid s) \mid r$	خاصیت شرکت‌پذیری	
$(r\ s)t = r(s\ t)$	خاصیت شرکت‌پذیری اپراتور الحاق
$r(s \mid t) = rs \mid rt$ یا $(s \mid t)r = sr \mid tr$	الحاق بر روی	توزیع‌پذیر است.
$r\varepsilon = r$ و $\varepsilon r = r$	ε عضو بی‌اثر الحاق می‌باشد.

هم عبارات منظم و هم گرامرهای منظم، قابلیت تعریف زبان‌های منظم را دارند. مثلاً، شناسه‌های زبان پاسکال توسط عبارات منظم زیر تعریف می‌شوند.

letter ← a|b|...|z

digit ← 0|1|...|9

id ← letter (letter | digit)*

به همین ترتیب، می‌توان گرامر منظم زیر را برای تعریف شناسه‌های زبان پاسکال به کار برد:

$S \rightarrow aA \mid bA \mid ... \mid zA \mid a \mid ... \mid z$

$A \rightarrow aA \mid bA \mid .... \mid zA \mid 0A \mid \mid A \mid .... \mid 9A \mid a \mid ... \mid z \mid 0 \mid .... \mid 9$

تفاوت دو تعریف فوق برای شناسه‌ها در این است که استفاده از عبارات منظم باعث راحتی از نظر نمایش، خوانایی بیش‌تر و فشرده‌تر شدن نمایش خواهد بود.

برای راحتی بیش‌تر، در بیان عبارت منظم، از دو علامت+ و ? نیز استفاده می‌گردد که به‌صورت زیر تعریف می‌شوند: در صورتی که r یک عبارت منظم باشد که زبان $L(r)$ را توصیف می‌کند، در آن صورت $r^+ = rr^*$ نیز یک عبارت منظم است و زبان $(L(r))^+$ را توصیف می‌نماید. اپراتور + دارای تقدم یکسانی با اپراتور * است. اگر r یک عبارت منظم باشد، در این صورت ?r یک عبارت منظم است که زبان $L(r) \cup \{\varepsilon\}$ را توصیف می‌کند.

هم‌چنین عبارت منظم $a \mid b \mid ... \mid z$ را می‌توان به صورت $[a - z]$ نمایش داد. به این ترتیب، می‌توان شناسه‌های زبان پاسکال را به صورت زیر نوشت:

$letter \leftarrow [a - z]$

$digit \leftarrow [0 - 9]$

$id \leftarrow letter\,(letter \mid digit)^*$

زبان‌های غیرمنظم (Nonregular Set) : برخی از زبان‌ها دارای توکن‌هایی هستند که نمی‌توان آن‌ها را با استفاده از عبارات منظم تعریف کرد. به عنوان مثال، پرانتزهای تودرتو را نمی‌توان به‌وسیله عبارات منظم تعریف کرد. این قبیل رشته‌ها را می‌توان با استفاده از گرامر مستقل از متن تعریف کرد. مجموعه $\{wcw^R \mid w \text{ is in } (a \mid b)^*\}$ را نه با کمک عبارات منظم و نه به‌وسیله گرامر مستقل از متن، نمی‌توان توصیف نمود.

تشخیص توکن‌ها

برای پیاده‌سازی اسکنر یک زبان دو روش وجود دارد: روش خودکار و روش دستی. برای تولید خودکار اسکنر یک زبان توسط نرم‌افزاری از قبیل Lex ، ابتدا باید فرم کلی توکن‌های آن زبان را توسط عبارات منظم، توصیف کرد. گرامر زیر را در نظر بگیرید، که در آن Expr ، Stmt و Term غیرپایانه و سایر علایم پایانه هستند.

Stmt → if Expr then stmt
| if Expr then Stmt else Stmt
| ε

Expr → Term relop Term
| Term

Term → id
| num

در گرامر فوق، فرم کلی پایانه‌ها به صورت زیر تعریف می‌شود:

if ← if

then ← then

else ← else

relop ← < | <= | = | < > | > | >=

id ← letter (letter | digit)*

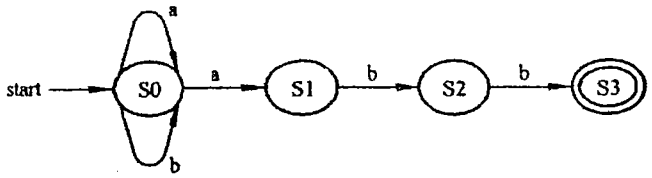
num ← digit+ (. digit+)? (E(+|-)?digit+)?

digit ← [0-9]

letter ← [a-z]

دیاگرام‌های انتقال

برای پیاده‌سازی دستی یک اسکنر، از ابزاری به نام دیاگرام انتقال (Transition Diagram) کمک می‌گیریم. یک دیاگرام انتقال در واقع یک گراف جهت‌دار است که هر یک از گره‌های آن، معرف یک وضعیت (State) است. یکی از وضعیت‌ها به عنوان وضعیت شروع و یکی (یا چند تا) از آن‌ها به عنوان وضعیت(های) خاتمه مشخص می‌گردد. برچسب (Label) هایی روی لبه‌های یک دیاگرام انتقال، قرار داده می‌شود که مشخص می‌کند در چه صورتی می‌توان از یک وضعیت به وضعیت دیگر رفت. هر دیاگرام انتقال، معرف یک زبان است. با خواندن کارکترهای یک رشته، تطبیق آن‌ها با برچسب‌های دیاگرام انتقال، معرف یک زبان و پیمایش آن دیاگرام می‌توان مشخص نمود که آیا آن رشته متعلق به زبان موردنظر است یا خیر. به عنوان مثال، دیاگرام انتقال زیر و عبارت منظم (a|b)* abb هر دو زبانی را توصیف می‌کنند که شامل رشته‌های تشکیل شده از علایم "a" و "b" که به زیر رشته "abb" ختم می‌شوند، است.

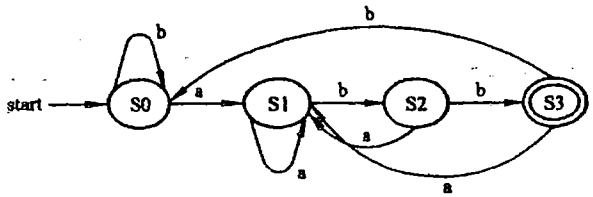


دیاگرام فوق یک دیاگرام نامعین (Nondeterministic) است. یک دیاگرام انتقال نامعین، دیاگرامی است که یکی از دو خاصیت زیر داشته باشد:

لبه‌های خارج شده از برخی از وضعیت‌های آن، بر چسب مشابه داشته باشند؛ و یا یکی از لبه‌های خارج شده از یک وضعیت، دارای برچسب ε باشد. (برچسب ε به این معنی است که بدون توجه به ورودی می‌توانیم از یک وضعیت به وضعیتی دیگر برویم.)

دیاگرام فوق، نامعین است زیرا از وضعیت S_0 دو لبه با برچسب مشترک "a" خارج شده است.

در صورتی که دیاگرامی نامعین نباشد، آن را معین (Determenistic) گویند. هم‌چنین هر دیاگرامی نامعینی را می‌توان به یک دیاگرام معین معادل تبدیل کرد. مثلاً، دیاگرام معین زیر، معادل دیاگرام فوق، است:



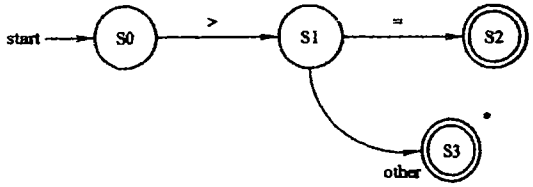
برنامه‌های که از یک دیاگرام معین استفاده می‌کند، پیاده‌سازی راحت‌تری نسبت به برنامه مبتنی بر یک دیاگرام نامعین دارد. برنامه مبتنی بر یک دیاگرام نامعین، بایستی دارای قابلیت پی‌جویی (Backtracking) باشد. از طرف دیگر، دیاگرام‌های انتقال معین معمولاً تعداد وضعیت بیش‌تری نسبت به دیاگرام نامعین معادل خود دارند.

بنابراین، برای پیاده‌سازی یک اسکنر ابتدا، دیاگرام‌های انتقال معرف الگوی توکن‌های زبان موردنظر رسم می‌گردد. از این دیاگرام‌ها برای به‌دست آوردن اطلاعات در مورد کارکترهایی که به‌وسیله نشانه‌رو Forward در ورودی، باید دیده شوند، استفاده می‌گردد. به این ترتیب که همان طوری‌که کارکترهای ورودی خوانده می‌شوند از یک وضعیت در دیاگرام به وضعیتی دیگر حرکت می‌کنیم، تا این‌که به یک وضعیت نهایی برسیم. پیمایش دیاگرام از وضعیت شروع (Start) آغاز می‌شود.

هنگامی که در وضعیت فعلی، لبه‌ای که برچسب آن مساوی کارکتر ورودی باشد، از آن حالت، به‌وسیله آن لبه به حالت بعدی می‌رویم و در غیر این صورت توکن، توسط این دیاگرام قابل تشخیص نخواهد بود.

برچسب "other" در روی لبه یک وضعیت بیان‌گر هر کارکتری است که توسط لبه‌های دیگر آن وضعیت ذکر نشده است.

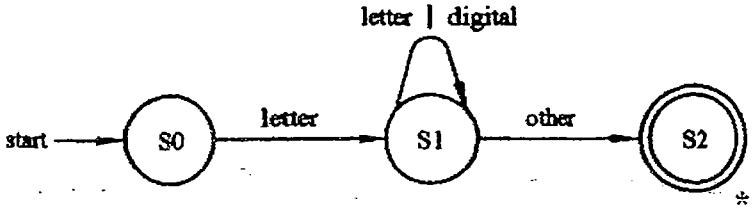
مثال : دیاگرام تشخیص توکن‌های = "<" و "<" و به صورت زیر است:



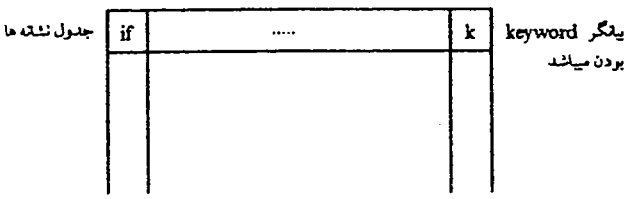
علامت ستاره، یعنی این که بایستی آخرین نویسه ورودی به میان‌گیر باز گردد.

در صورتی که در کلیه دیاگرام‌ها نتوان یک توکن را تشخیص داد، یک خطای واژه‌ای روی داده است و باید برنامه خطاپرداز، فراخوانی شود.

مثال : دیاگرام تشخیص شناسه‌های زبان پاسکال



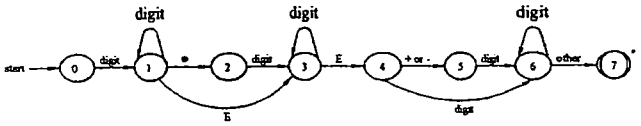
با توجه به این که کلمات کلیدی از یک دنباله کارکتری تشکیل شده‌اند، می‌توان از دیاگرام فوق، برای تشخیص کلمات کلیدی نیز استفاده نمود. تشخیص کلمات کلیدی و شناسه‌ها، توسط یک دیاگرام، باعث کاهش تعداد وضعیت‌های دیاگرام انتقال اسکنر می گردد. برای آن‌که کلمات کلیدی را از شناسه‌های هم نامشان جدا سازیم، یکی از ساده‌ترین روش‌ها این است که در ابتدا در جدول نشانه‌ها کلمات کلیدی را وارد کنیم. به این ترتیب با رجوع به جدول نشانه‌ها می‌توان دریافت که توکن موردنظر شناسه است یا کلمه کلیدی.

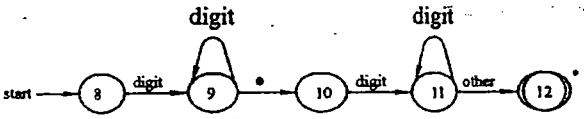


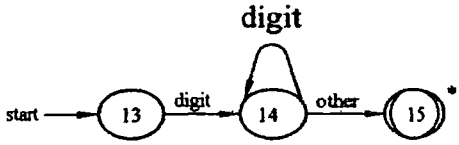
برای این کار از دو تابع ( )gettoken و ( ) id - install استفاده می‌شود . تابع id – install جدول علایم را جستجو می‌کند و اگر واژه توکن در جدول نشانه‌ها به عنوان کلمه کلیدی آمده باشد، این تابع عدد صفر را باز می‌گرداند. در صورتی که واژه یک متغیر، تشخیص داده شود و در جدول هم موجود باشد، این تابع، آدرس آن متغیر در جدول نشانه‌ها را به وسیله یک اشاره‌گر باز می‌گرداند. اگر چنان‌چه واژه در جدول موجود نباشد، به عنوان ورودی جدید به جدول علایم وارد شده و مطابق حالت قبل، آدرس آن بازگردانده خواهد شد.

( )Gettoken نیز به‌طور مشابهی، جدول نشانه‌ها را جستجو کرده، در صورتی که واژه موردنظر یک کلمه کلیدی باشد، توکن متناظرش را مستقیماً به پارسر می‌فرستد و در غیر این صورت توکن "id" را انتقال می‌دهد. به این ترتیب، در صورتی که تعداد کلمات کلیدی تغییر کند، دیاگرام انتقال بدون تغییر باقی خواهد ماند و به راحتی می‌توان این تغییر را در جدول نشانه‌ها ایجاد کرد.

در این‌جا ذکر چند نکته در مورد نحوه قرار دادن دیاگرام انتقال توکن‌های مختلف ضروری است. اول آن‌که اسکنر باید همواره سعی کند طولانی‌ترین توکن ممکن را تشخیص دهد؛ مثلاً دیاگرام تشخیص اعداد اعشاری بایستی قبل از دیاگرامی باشد که اعداد صحیح را تشخیص می‌دهد. هم‌چنین دیاگرام تشخیص توکن‌هایی که مورد استفاده بیش‌تری در برنامه‌ها دارند (مثلاً tab,space و غیره)، باید قبل از دیاگرام انتقال توکن‌های کمیاب‌تر قرار گیرد تا در نهایت تست کمتری برای تشخیص توکن‌ها انجام شود. مثلاً شکل زیر ترتیب، قرار گرفتن دیاگرام‌ها برای تشخیص اعداد را نشان می‌دهد. شماره وضعیت‌ها بیان‌گر ترتیب دیاگرام‌ها است.







بعد از این که دیاگرام تشخیص توکن‌ها رسم شد، به راحتی می‌توان آنرا با دستور case پیاده‌سازی نمود.

هر دیاگرام انتقال نامعین را می‌توان به یک گرامر مستقل از متن تبدیل کرد. برای این کار باید مراحل زیر را انجام داد:

۱ـ به ازای هر حالت i یک غیرپایانه Ai در نظر می‌گیریم.

۲ـ اگر از حالت i با ورودی a به حالت j می‌رویم، قاعده‌ای به‌صورت Ai → a Aj تولید می‌کنیم.

۳ـ اگر از حالت i با ورودی ε به حالت j می‌رویم، قاعده‌ای به فرم Ai → Aj تولید می‌کنیم.

۴ـ اگر i یک حالت نهایی باشد، قاعده‌ای به فرم Ai → ε تولید می‌کنیم.

۵ـ اگر i حالت شروع باشد ، غیر پایانه Ai را به عنوان علامت شروع گرامر در نظر می‌گیریم.

مثلاً گرامر مستقل از متن دیاگرام انتقال نامعین $(a \mid b)^* abb$ به صورت زیر خواهد بود:

$A_0 \rightarrow aA_1 \mid aA_0 \mid bA_0$
$A_1 \rightarrow b A_2$
$A_2 \rightarrow b A_3$
$A_3 \rightarrow \varepsilon$

اگرچه می‌توان قواعد لغوی زبان‌ها را توسط گرامرهای مستقل از متن نیز بیان نمود. لیکن، دلایلی وجود دارد که بهتر است قواعد لغوی توسط عبارت منظم توصیف شوند:

۱ـ قواعد لغوی زبان‌ها اغلب، خیلی ساده هستند و برای توصیف آن‌ها، نیازی به نمایش قوی‌تر گرامر مستقل از متن نیست.

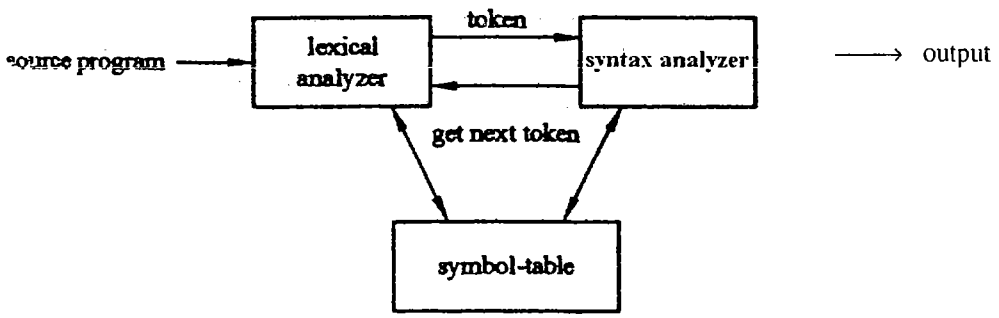
۲ـ عبارات منظم به‌طور کلی، وسیله‌ای فشرده‌تر و گویاتر از گرامرهای مستقل از متن هستند.

۳ـ اسکنرهای سریع‌تری را می‌توان به‌صورت خودکار از روی عبارات منظم تولید کرد.

۴ـ جدا کردن ساختار دستوری یک زبان به دو بخش لغوی و غیرلغوی، کار پیاده‌سازی قسمت Front –End کامپایلرها را به صورت پیمانه‌ای راحت‌تر می‌سازد.

تحلیل نحوی (Syntax Analysis)

در مرحله تحلیل نحوی، برنامه ورودی از نظر دستوری بررسی می‌شود. تحلیل‌گر نحوی یا پارسر، برنامه ورودی را که به صورت دنباله‌ای از توکن‌ها است، از اسکنر گرفته و تعیین می‌کند که آیا این جمله می‌تواند به‌وسیله گرامر زبان موردنظر تولید شود یا خیر؟ رابطه پارسر و اسکنر به صورت زیر است؟



به‌طور کلی سه نوع روش تحلیل نحوی وجود دارد:

۱ـ روش عمومی (Universal) که چندان کارا نیست، ولی با هر نوع گرامری کار می‌کند.

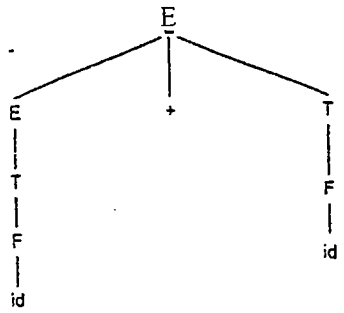
۲ـ روش‌های بالا به پایین (Top – Down)

۳ـ روش‌های پایین به بالا (Bottom – Up)

روش‌های بالا به پایین پارس، درخت تجزیه (Parse Tree) را از بالا به پایین می‌سازند، در حالی‌که روش‌های پایین به بالا، برعکس، عمل می‌کنند؛ یعنی درخت تجزیه را از پایین به بالا تولید می‌کنند. در هر دو روش، ورودی از چپ به راست و در هر قدم فقط یک توکن بررسی می‌شود. به عنوان مثال، گرامر زیر را در نظر بگیرید:

1,2 $E \rightarrow E+T \mid T$
3,4 $T \rightarrow T*F \mid F$
5,6 $F \rightarrow (E) \mid id$

درخت تجزیه جمله id+id به‌صورت زیر خواهد بود. که در آن ، ریشه درخت، علامت شروع گرامر است. (توضیح آن‌که از این پس اگر علامت شروع گرامری به‌صورت صریح مشخص نشود، غیرپایانه سمت چپ، اولین قاعده گرامر به عنوان علامت شروع آن در نظر گرفته می‌شود.)



مهم‌ترین روش‌های تجزیه بالا به پایین عبارتند از:

۱ـ روش پایین‌گرد (Recursive Descent)

۲ـ روش LL(1) که حالت خاصی از روش کلی LL(k) است.

در روش LL(k) منظور از 'L' اول این است که ورودی از سمت چپ به راست خوانده و بررسی می‌گردد، 'L' دوم یعنی پارسر از بسط چپ (Left – Most Derviation) استفاده می‌کند و k نیز بیان‌گر این است که در هر قدم از تجزیه، فقط یک توکن بررسی خواهد شد.

مهم‌ترین روش‌های تجزیه پایین به بالا عبارتند از:

۱ـ روش تقدم عملگر (Operator Precedence)

۲ـ روش تقدم ساده (Simple Precedence)

۳ـ روش LR(1) که خود دارای سه فرم مختلف با نام‌های SLR(1) , LALR(1) , CLR(1) است.

تمامی روش‌های فوق به نوعی از یک روش کلی به نام انتقال ـ کاهش (Shift – Reduce) پیروی می‌کنند. اگرچه روش‌های تجزیه بالا به پایین برای پیاده‌سازی دستی مناسب‌ترند، اما ابزارهایی وجود دارد که با کمک آن‌ها می‌توان به‌طور خودکار پارسرهای قوی پایین به بالا تولید نمود. در LR(1) منظور از 'L' یعنی پارسر ورودی را از چپ به راست می‌خواند و منظور از R این است که پارسر از عکس بسط راست (Right-Most Derivation) استفاده می‌کند.

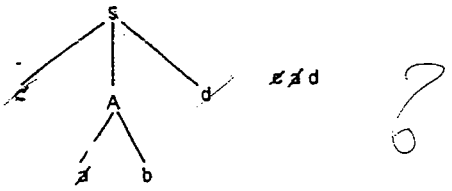
تجزیه بالا به پایین (Top – Down Parsing)

در حالت کلی، یک پارسر بالا به پایین بایستی بتواند در صورت لزوم عمل پی‌جویی (Back Tracking) انجام دهد. گرامر زیر را در نظر بگیرید.

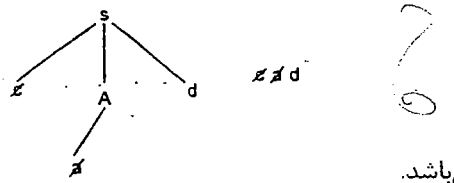
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

برای تجزیه رشته ورودی cad، پارسر به‌صورت زیر عمل می‌کند.



چون d با b برابر نیست، پارسر نیاز دارد که یک مرحله به عقب باز گردد و قاعده دیگر A را مورد بررسی قرار دهد.



در این جا پارسر نتیجه می‌گیرد که رشته ورودی قابل قبول می‌باشد.

اگرچه همان‌گونه که توضیح داده شد، عمل تحلیل نحوی در حالت کلی می‌تواند به روش آزمایش و خطا اجرا گردد، لیکن بهتر است پارسرها، به گونه‌ای طراحی و پیاده‌سازی شوند که نیازی به پی‌جویی نداشته باشند. به پارسرهایی که عمل پی‌جویی انجام نمی‌دهند، پارسر پیشگو (Predictive) می‌گویند از طرفی پارسرها معمولاً به صورت حریصانه (Greedy) عمل می‌کنند. یعنی با دریافت هر توکن، درخت تجزیه را تا حد امکان گسترش می‌دهند و تنها هنگامی که دیگر امکان گسترش درخت پارس وجود نداشته باشد، توکن بعدی را درخواست می‌کنند.

به عنوان مثال، گرامر زیر را که معرف گونه (Type) در زبان پاسکال است، در نظر بگیرید.

$Type \rightarrow Simple$

$\mid \uparrow id$

$\mid array\ [Simple]\ of\ Type$

$Simple \rightarrow integer$

$\mid char$

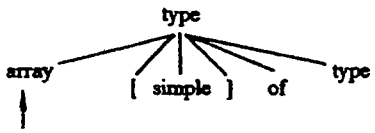
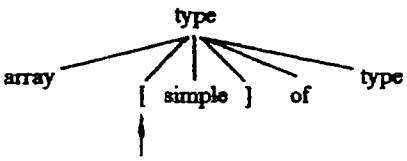
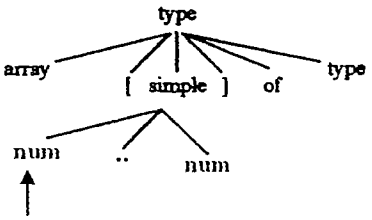
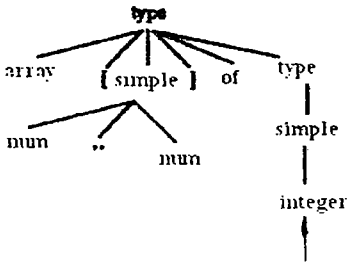
$\mid num..num$

در روش‌های بالا به پایین تولید، درخت تجزیه از ریشه درخت که همان غیرپایانه شروع گرامر است، آغاز و در ادامه کار، مراحل زیر به‌طور مکرر انجام می‌گردد و درخت تجزیه به‌صورت بالا به پایین و از چپ به راست ساخته می‌شود.

۱ـ در گره n با غیرپایانه A، یکی از قواعد گرامر که A در سمت چپ آن قرار دارد را انتخاب کرده و سمت راست این قاعده را به عنوان فرزندان گره n درخت پارس قرار می‌دهد.

۲ـ گره بعدی را که از آن‌جا یک زیر درخت دیگر باید ایجاد شود، پیدا می‌کند.

مراحل فوق، در حین آن‌که رشته ورودی از چپ به راست خوانده می‌شود، انجام می‌گیرد. توکنی که پارسر در حال بررسی آن است را توکن جاری گویند. فرض کنید رشته ورودی به صورت "array [num .. num] of integer" باشد. مراحل تشکیل درخت پارس به صورت زیر خواهد بود.

وضعیت ورودی	درخت تجزیه
array [num .. num] of integer ↑	
array [num .. num] of integer ↑	
array [num .. num] of integer ↑	
arrat [num .. num] of integer ↑	

تجزیه پایین‌گرد (Recursive Descent Parsing)

یکی از انواع پارسرها که به صورت پیش‌گویانه عمل می‌کند، پارسر پایین‌گرد (Recursive Descent) است. این پارسر به صورت بالا به پایین عمل می‌کند و در آن، یک مجموعه رویه به‌طور بازگشتی رشته ورودی را مورد پردازش قرار می‌دهند. این رویه‌ها که برای پردازش رشته فراخوانی می‌شوند، یک درخت پارس برای ورودی ایجاد می‌کنند. یک پارسر پایین‌گرد به ازای هر غیر پایانه یک رویه دارد که دو کار انجام می‌دهد:

۱ـ تصمیم می‌گیرد که از کدام قاعده آن غیرپایانه استفاده شود.

۲ـ از قاعده انتخاب شده استفاده می‌کند.

علاوه بر رویه‌هایی که به ازای هر غیر پایانه وجود دارد. یک پارسر پایین‌گرد از رویه دیگری به نام match برای تطبیق توکن‌های ورودی و پایانه‌های درخت تجزیه در حال ساخت استفاده می‌کند. به عنوان مثال، پارسر پایین‌گرد برای گرامر تعریف Type در زبان پاسکال دو رویه برای غیر پایانه‌های Type و Simple خواهد داشت، رویه Match نیز به‌صورت زیر است:

```
procedure match (t:token);
begin
    if lookahead=t Then
        lookahead : = nexttoken
    else error
end
```

رویه فوق، برای راحتی کار رویه‌های Type و Simple استفاده می‌شود و متغیر Lookahead را که حاوی توکن جاری است، تغییر می‌دهد.

```
procedure Type;
begin
        if lookahead is in {integer , char, num} Then Simple
       else if lookahead = '↑'
       Then begin
        match ('↑');
       match ( id ) ;
       end
        else if lookahead is array Then begin
            match ('[' )
             match (array);
             Simple;
             match ( ' ] ' )
             match(of);
             Type
          end
          else error
end
procedure Simple;
begin
          if lookahead= integer Then match (integer)
          else if lookahead = char Then match (char)
          else if lookahead= num Then begin
               match (num); match ( .. ); match (num)
          end
          else error
end
```

پارسر پایین‌گرد با استفاده از حاصل اعمال تابعی به نام "First" بر روی رشته سمت راست قواعد، تعیین می‌کند که از کدام یک از قواعد گرامر باید استفاده شود. تابع First روی رشته‌ای از پایانه‌ها و غیرپایانه‌ها عمل می‌کند. حاصل تابع $First(\alpha)$ مجموعه‌ای از پایانه‌ها است که در سمت چپ‌ترین قسمت از رشته‌های تولید شده از رشته α قرار می‌گیرند. تعریف رسمی‌تر این تابع به‌صورت زیر است:

$$First(\alpha) = \{a \mid \alpha \Rightarrow^{*} a\beta\}$$

الگوریتم به‌دست آوردن تابع First یک رشته در بخش‌های بعدی آورده شده است. به‌عنوان مثال گرامر زیر را در نظر بگیرید.

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

$First(cAd) = \{c\}$,
$First(S) = \{c\}$,
$First(A) = \{a\}$.

برای گرامر Type خواهیم داشت:

First (Simple) ={ integer . char . num}
First (↑ id)= {↑}
First (array [Simple] of Type)={ array}

به این ترتیب، در گرامری که دو قاعده به‌صورت $A \rightarrow \beta$، $A \rightarrow \alpha$ داشته باشد؛ پارسر پایین‌گرد با استفاده از First سمت راست این قواعد، قاعده مناسب را تعیین می‌کند؛ بدون این که نیازی به عمل عقب‌گرد داشته باشد. البته مشروط بر این که در چنین گرامرهایی شرط زیر برقرار باشد:

$$First(\alpha) \cap First(\beta) = \phi$$

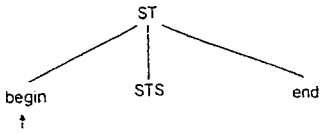
استفاده از قواعد اپسیلون $(A \rightarrow \varepsilon)$

هر گاه، در گرامری قاعده اپسیلون وجود داشته باشد، پارسر پایین‌گرد از آن به عنوان قاعده پیش فرض (Default) استفاده می‌کند، به این معنی که اگر هیچ قاعده دیگری در گرامر مناسب تشخیص داده نشده پارسر از قاعده اپسیلون برای ادامه عمل تجزیه استفاده می‌کند.

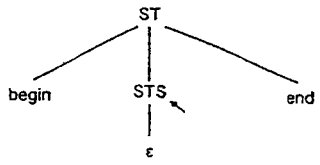
مثال :

$ST \rightarrow$ begin STS end
$STS \rightarrow STL \mid \varepsilon$

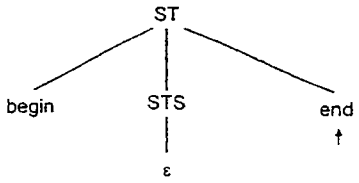
فرض کنید جمله ورودی begin end باشد. درخت تجزیه پس از خواندن توکن begin و بسط علامت شروع گرامر به‌صورت زیر خواهد بود:



پس از تطبیق توکن begin، توکن بعدی یعنی end از اسکنر دریافت می‌گردد و نوبت به بسط غیرپایانه STS می‌رسد. چون end متعلق به مجموعه First (STS) نیست، لذا از قاعده $STS \rightarrow \varepsilon$ به عنوان قاعده پیش‌فرض استفاده می‌شود. درخت تجزیه به‌صورت زیر در خواهد آمد:



سپس توکن end نیز با پایانه end در درخت تجزیه تطبیق می‌شود و عمل تجزیه خاتمه می‌یابد:



باید توجه داشت که انتخاب قاعده ε برای بسط STS تنها در صورتی که توکن جاری در آن لحظه end باشد، انتخاب درستی خواهد بود.

مشکل چپ‌گردی (Left Recursion)

گرامری را چپ‌گرد (Left Recursive) گویند، که غیرپایانه سمت چپ یک قاعده آن به عنوان اولین علامت سمت راست آن قاعده ظاهر شده باشد. به عبارتی دیگر، گرامر قاعده‌ای به‌صورت $A \rightarrow A\alpha$ داشته باشد.

روش‌های پارس بالا به پایین را نمی‌توان برای گرامری که چپ‌گردی داشته باشد، به‌کار برد. از این رو باید چپ‌گردی گرامر را حذف کنیم؛ یعنی گرامر را به گرامر معادلی تبدیل کنیم که در آن چپ‌گردی وجود نداشته باشد. برای مثال، گرامر چپ‌گرد $A \rightarrow A\alpha|\beta$ را می‌توان به فرم زیر که چپ‌گردی ندارد، تبدیل نمود:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A'|\varepsilon$$

هر دو گرامر فوق رشته‌هایی به فرم $\beta\alpha^*$ توصیف می‌کنند.

روش کلی حذف چپ‌گردی به‌صورت زیر است. توجه کنید که اهمیتی ندارد که چه تعداد از قواعد چپ‌گرد باشند. به‌طور کلی، اگر داشته باشیم:

$$A \rightarrow A\alpha_1|A\alpha_2|\ldots|A\alpha_m|\beta_1|\beta_2|\ldots|\beta_n$$

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

در قواعد فوق، فرض بر این است که βiها نباید با A شروع شوند و هیچ‌کدام از αi نباید ε باشند. در این صورت می‌توان قواعد زیر را به جای قواعد چپ‌گرد فوق به کار برد:

$A \to \beta_1 A' \mid \dots \mid \beta_n A'$

$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$

این گونه چپ‌گردی‌ها را "چپ‌گردی آشکار" (Explicit Left Recursion) گویند.

ممکن است، چپ‌گردی در بیش از یک قدم ظاهر شود، که به آن چپ‌گردی ضمنی گویند. به عنوان مثال، گرامر زیر دارای چپ‌گردی ضمنی است.

$$\begin{cases} S \to Aa \mid b \\ A \to Ac \mid Sd \end{cases}$$

در غیر پایانه S چپ‌گردی ضمنی داریم زیرا :

$S \Rightarrow Aa \Rightarrow Sda$

حذف چپ‌گردی ضمنی (Implicit Left Recursion)

ورودی الگوریتم گرامر G با این شرط که قاعده اپسیلون نداشته باشد و هیچ دوری نیز در گرامر موجود نباشد؛ یعنی بسطهایی به‌صورت $A \Rightarrow^{+} A$ در گرامر نباشد. خروجی الگوریتم، گرامری معادل گرامر G، اما فاقد چپ‌گردی است. ابتدا غیرپایانه‌های گرامر را به ترتیب دلخواه $A_1, A_2, \dots, An$ مرتب می‌کنیم. سپس اعمال زیر را به صورت مشخص شده در حلقه‌های تکرار اجرا می‌کنیم:

```
For i:= 1 to n do
  For j :=1 to i – 1 do begin
```

به‌جای هر قاعده به شکل $A_i \to A_j\gamma$ قواعد $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$ را قرار دهید،
که در آن $A_j \to \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ قواعد فعلی A_j هستند.

```
    end
end
```

سپس چپ‌گردی آشکار قواعد A_i را حذف کنید.

به عنوان نمونه، الگوریتم فوق را برای گرامر زیر به کار می‌بریم.

$S \to Aa \mid b$

$A \to Ac \mid Sd$

ابتدا، غیر پایانه‌های گرامر را به ترتیب S,A (از چپ به راست) مرتب می‌کنیم. سپس در صورتی که در قواعد اولین غیرپایانه (در این‌جا S) چپ‌گردی آشکار وجود داشته باشد، چپ‌گردی آشکار آن‌ها را برطرف می‌کنیم. در مرحله بعد از روی قاعده $A \to Sd$ خواهیم داشت:

$A \to Aad \mid bd$

به این ترتیب، قواعد A به‌صورت زیر در خواهند آمد که دارای چپ‌گردی آشکار هستند.

$A \to Aad$

$A \to Ac$

$A \to bd$

با حذف چپ‌گردی آشکار قواعد فوق، گرامر زیر به‌دست می‌آید.

$S \to Aa \mid b$
$A \to bd\ A'$
$A' \to cA' \mid adA' \mid \varepsilon$

فاکتورگیری از چپ(Left Factoring)

با استفاده از فاکتورگیری از چپ، می‌توان گرامرهایی که در آن‌ها برای غیرپایانه A دو قاعده به‌صورت $A \to \alpha\beta_1$ و $A \to \alpha\beta_2$ وجود دارد را طوری تغییر داد که بتوان پارس بالا به پایین را برای این گرامرها استفاده کرد.
مشکل این قبیل گرامرها در این است که روشن نیست که از کدام یک از قواعد باید برای بسط غیرپایانه A استفاده کرد.

مثال :

stmt → if exp then stmt else stmt
| if exp then stmt

با دیدن if در ورودی، بلافاصله نمی‌توان گفت که از کدام قاعده برای بسط غیرپایانه stmt می‌توان استفاده کرد.
در حالت کلی، اگر $A \to \alpha\beta_1 \mid \alpha\beta_2$ دو قاعده موجود در گرامری باشند و ورودی با رشته α شروع شده باشد، نمی‌توان گفت که A را باید به‌صورت $\alpha\beta_1$ بسط داد و یا به‌صورت $\alpha\beta_2$. برای رفع این مشکل از α فاکتور می‌گیریم و گرامر را به‌صورت زیر تبدیل می‌کنیم:

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

الگوریتم فاکتورگیری از چپ در حالت کلی به‌صورت زیر است:

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \delta_1 \mid \delta_2 \mid \ldots \mid \delta_n$

که در آن، δ_i بیان‌گر قواعدی است که سمت راست هیچ کدام با α آغاز نشده است. با فاکتورگیری از چپ قواعد زیر حاصل می‌شوند:

$A \to \alpha A' \mid \delta_1 \mid \delta_2 \mid \ldots \mid \delta_n$
$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

که در آن A' یک غیرپایانه جدید است.

مثال : قواعد گرامری به‌صورت زیر می‌باشد:

$S \to iEtS$
$S \to \underbrace{iEtSeS}_{s'} \mid a$
$E \to b$

این قواعد بعد از انجام عمل فاکتورگیری از چپ به صورت زیر تبدیل می‌شوند:

$S \to iEtSS' \mid a$
$S' \to eS \mid \varepsilon$
$E \to b$

زبان‌های غیر مستقل از متن (Non Context Free language)

زبان‌هایی هستند که نمی‌توان آن‌ها را توسط یک گرامر مستقل از متن توصیف کرد. هم‌چنین محدودیت‌هایی در زبان‌های برنامه نویسی وجود دارد که نمی‌توان آن‌ها را توسط گرامرهای مستقل از متن اعمال کرد. به مثال‌های زیر توجه کنید:

مثال ۱ : زبان زیر را در نظر بگیرید:

$L_1 = \{wcw \mid w \text{ is in } (a \mid b)^*\}$

این زبان، رشته‌هایی به صورت aabcaab تولید می‌کند. این رشته‌ها را می‌توان مشابه این محدودیت، در زبان‌های برنامه‌سازی در نظر گرفت که تعریف متغیرها بایستی قبل از استفاده از آن‌ها قرار گیرد؛ به این ترتیب که w اول در wcw بیان‌گر تعریف متغیر بوده و w دوم نشان‌دهنده استفاده از متغیر می‌باشد.

declaration $\frac{abc}{w}$

```
begin
 c
end
```

$\frac{abc}{w} \ldots$

مثال ۲ : زبان $L_2 = \{a^n b^m c^n d^m \mid m \geq 1 \text{ and } n \geq 1\}$ نیز مستقل از متن نیست. این زبان رشته‌هایی به صورت $a^+b^+c^+d^+$ تولید می‌کند که در آن‌ها تعداد تکرار a با c برابر است و تعداد تکرار b با d برابر است. این زبان، مشابه این محدودیت در زبان‌های برنامه‌سازی است که تعداد پارامترها رسمی در تعریف یک رویه بایستی با تعداد آرگومان‌ها در فراخوانی همان رویه برابر باشد. در این‌جا می‌توان a^n و b^m را بیان‌گر پارامترهای رسمی در تعریف دو رویه که به ترتیب دارای n و m پارامتر ورودی هستند در نظر گرفت؛ به همین ترتیب، d^m و c^n را می‌توان به عنوان تعداد آرگومانها در فراخوانی این رویه‌ها در نظر گرفت.

```
dcl    proc1 (a,a,a)
dcl    proc2 (b,b)
⋮
call   proc1 (c,c,c)
call   proc2 (d,d)
```

مثال ۳ : زبان $L_3 = \{a^n b^n c^n \mid n \geq 0\}$ نیز مستقل از متن نیست. این زبان، رشته‌هایی به فرم $a^+b^+c^+$ تولید می‌کند که در آن‌ها تعداد a،b و c برابر است. این زبان مشابه مسأله ایجاد کلماتی که در زیر آن‌ها خط کشیده شده باشد (Underlined Word). این گونه کلمات، این‌گونه چاپ می‌شوند که ابتدا، کارکترهای یک کلمه چاپ شده، به‌دنبال آن به تعداد کارکترهای آن کلمه به عقب برگشته (با کمک کاراکتر back space) و سپس به همان تعداد کاراکتر "-" چاپ می‌شود. مثلاً کلمه Read در زبان L_3 اگر a بیان‌گر حروف ، b بیان‌گر کاراکتر bakspace و c نیز بیان‌گر کارکتر "-" باشد، آن‌گاه این زبان کلمات Underline را تولید می‌کند.

گرامرهایی وجود دارند که با وجود شباهت بسیار به گرامرهای L_1، L_2 و L_3 در مثال‌های فوق، مستقل از متن هستند. در مثال زیر: L_1 که به صورت زیر تعریف شده، مستقل از متن است.

$L_1' = \{wcw^R \mid w \text{ is in } (a \mid b)^*\}$

این زبان را می‌توان به‌وسیله گرامر زیر تولید کرد.

$S \rightarrow aSa \mid bSb \mid c$

زبان $L_2' = \{a^n b^n c^m d^m \mid n \geq 1\}$ نیز مستقل از متن است و با گرامر زیر توصیف می‌شود:

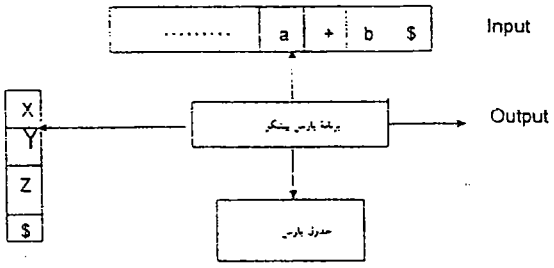
$S \rightarrow AB$
$A \rightarrow aAb \mid ab$
$B \rightarrow cBd \mid cd$

و سرانجام، زبان $L_3' = \{a^n b^n \mid n \geq 1\}$ نیز مستقل از متن و گرامر تولیدکننده آن به‌صورت زیر است:

$S \rightarrow aSb \mid ab$

تجزیه پیش‌گویانه غیربازگشتی (Non Recursive Predictive Parsing)

در این روش تجزیه، از یک انباره (Parse Stack) و یک جدول بنام جدول تجزیه (Parsing Table) استفاده می‌گردد. میان‌گیر ورودی شامل رشته‌ای است که باید تجزیه شود. در انتهای رشتهٔ ورودی علامتی مثلا $ قرار می‌گیرد. ساختار کلی این نوع پارسر به فرم زیر است:



در ابتدای پارس، علامت $ وارد انباره می‌شود و روی آن، علامت شروع گرامر قرار می‌گیرد. در انتهای پارس، هم در انباره و هم در ورودی تنها علامت $ باقی می‌ماند. جدول پارس، یک آرایهٔ دو بعدی به‌صورت M[A,a] است که در آن A یک غیرپایانه و a یک پایانه و یا علامت $ است. فرم کلی جدول به صورت زیر است:

	پایانه‌ها و علامت $			
غیر پایانه‌ها			.....	

برخی از خانه‌های جدول، حاوی شماره یک قاعده از گرامر و برخی از آن‌ها خالی است.

در هر قدم از پارس، برنامهٔ پارس علامت X بالای انباره و توکن جاری a در ورودی را مورد بررسی قرار می‌دهد و به‌صورت زیر تصمیم می‌گیرد:

۱ـ اگر $X = a = \$$ باشد، پارسر پایان موفقیت‌آمیز پارس را گزارش می‌کند.

۲ـ اگر $X = a \neq \$$ باشد، پارسر X را از بالای انباره، حذف و توکن بعدی را دریافت می‌کند. اگر X پایانه باشد و با a مطابقت نکند، یک خطای نحوی رخ داده است.

۳ـ اگر X یک غیرپایانه باشد، برنامه به خانه M[X,a] مراجعه می‌کند که در آن یا شماره قاعده‌ای به فرم $A \rightarrow XYZ$ قرار دارد و یا خالی است. در صورت اول، پارسر X را از بالای انباره حذف و به‌جای آن XYZ را وارد انباره می‌کند. به نحوی که X بالای انباره قرار گیرد. در صورتی که خانه M[X,a] خالی باشد، یک خطای نحوی رخ داده است.

توابع First و Follow

برای پر کردن جدول پارس از توابعی با نام‌های First و Follow استفاده می‌شود. همان‌گونه که در قبل توضیح داده شد، First (α) مجموعهٔ پایانه‌هایی است که به عنوان سمت چپ‌ترین علامت رشته‌های به‌دست آمده از α قرار می‌گیرند. در صورتی که $\alpha \Rightarrow^* \varepsilon$ در این صورت ε نیز جزو First (α) خواهد بود. در ادامه الگوریتم محاسبه First یک علامت مثل X توضیح داده می‌شود.

اگرچه الگوریتم در مورد یک علامت بیان می‌گردد، لیکن با کمک آن می‌توان مجموعه First را برای رشته‌ها نیز محاسبه نمود.

برای پیدا کردن First (X) به صورت زیر عمل می‌شود (X می‌تواند یک پایانه و یا یک غیرپایانه باشد).

۱ـ اگر X یک پایانه باشد در آن‌صورت $First(X)=\{X\}$.

۲ـ اگر قاعده‌ای به‌صورت $X \to \varepsilon$ در گرامر باشد، ε را به First(X) اضافه می‌کنیم.

۳ـ اگر قاعده‌ای به فرم $X \to Y_1Y_2...Y_k$ در گرامر موجود باشد، ابتدا $First(Y_1)-\varepsilon$ (یعنی مجموعه $First(Y_1)$ منهای ε) را به First(X) اضافه می‌کنیم. در صورتی که $First(Y_2)-\varepsilon . Y_1 \Rightarrow^* \varepsilon$ ، را نیز به First(X) می‌افزاییم. (در غیر این صورت کار یافتن First(X) از طریق قاعده فوق خاتمه می‌یابد). این کار آن‌قدر ادامه می‌یابد تا آن‌که ε عضو مجموعه First(Yi) به ازای $0 \le i < k$، نباشد. در صورتی که $Y_{k-1} \Rightarrow^* \varepsilon$ ، باید $First(Y_k)$ را نیز به First (X) اضافه کنیم.

به عنوان مثال، گرامر زیر را در نظر بگیرید:

$A \to aB$

$B \to Cb \mid d$

$C \to \varepsilon \mid c$

$First(A)=\{a\}. First(B)=\{c, d, b\}$. $First(C)=\{\varepsilon, c\}$

برای به‌دست آوردن Follow(A) (A یک غیرپایانه است) اعمال زیر را آن‌قدر ادامه می‌دهیم تا اینکه دیگر چیزی به مجموعه Follow (A) اضافه نشود.

۱ـ $ را در Follow (S) قرار می‌دهیم (S علامت شروع گرامر است).

۲ـ اگر قاعده‌ای به‌صورت $X \to \alpha A \beta$ داشته باشیم، هر چه در First (β) قرار دارد (بغیر از ε) را به مجموعهٔ Follow (A) اضافه می‌کنیم.

۳ـ اگر قاعده‌ای به فرم $X \to \alpha A$ داشته باشیم و یا آن‌که قاعده‌ای به فرم $X \to \alpha A \beta$ و $\beta \Rightarrow^* \varepsilon$ ، هر چه در Follow (X) قرار دارد را به مجموعهٔ Follow (A) اضافه می‌کنیم.

به عنوان مثال، گرامر زیر را در نظر بگیرید:

1 $E \to TE'$

2-3 $E' \to +TE' \mid \varepsilon$

4 $T \to FT'$

5-6 $T' \to *FT' \mid \varepsilon$

7-8 $F \to (E) \mid id$

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

مجموعه‌های Follow	مجموعه‌های First
Follow (E) = {),$}	First (E) = {(,id}
Follow (T) = {+,),$}	First (T) = {(,id}
Follow (F) = {*,+,$,)}	First (F) = {(,id}
Follow (E') = {), $}	First (E') = {ε,+}
Follow (T') = {+,),$}	First (T') = {ε,*}

نحوهٔ تشکیل جدول پارس، پارسرهای پیش‌گو:

۱ـ برای هر قاعده به‌صورت $A \to \alpha$ گرامر، قدم‌های ۲ و ۳ را انجام می‌دهیم.

۲ـ برای هر پایانه a در $First(\alpha)$، شماره قاعده $A \to \alpha$ را به خانه M[A,a] اضافه می‌کنیم.

۳ـ اگر ε در $First(\alpha)$ وجود داشت، شماره قاعده $A \to \alpha$ را در خانه‌های M[A,b] به ازای هر $b \in Follow(A)$ قرار می‌دهیم.

جدول پارس گرامر مثال قبل، به فرم زیر است.

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

۱/ $E \to TE'$

۲،۳/ $E' \to +TE' \mid \varepsilon$

۴/ $T \to FT'$

۵،۶/ $T' \to *FT' \mid \varepsilon$

۷و۸/ $F \to (E) \mid id$

حال با استفاده از جدول فوق عبارت id+id*id را تجزیه می‌کنیم:

محتوای انباره	ورودی	قواعد استفاده شده
$E	id+id*id$	E → TE'
$E'T	id+id*id$	T → FT'
$E'T'F	id+id*id$	F → id
$E'T'id	id+id*id$	
$E'T'	+id*id$	T' → ε
$E'	+id*id$	E' → +TE'
$E'T+	+id*id$	
$E'T	id*id$	T → FT'
$E'T'F	id*id$	F → id
$E'T'id	id*id$	
$E'T'	*id$	T' → *FT'
$E'T'F*	*id$	
$E'T'F	id$	F → id
$E'T'id	id$	
$E'T'	$	T' → ε
$E'	$	E' → ε
$	$	پایان پارس

گرامرهای LL(1): درصورتی که از روش فوق برای ایجاد جدول پارس گرامرهای گنگ و یا چپ‌گرد استفاده شود، در برخی از خانه‌های جدول پارس بیش از یک شماره قاعده خواهیم داشت. به عبارت دیگر، اگر در خانه‌های جدول پارس یک گرامر مستقل از متن حداکثر یک شماره قاعده باشد، گرامر مربوطه را LL(1) گویند.

مثال: گرامر زیر را در نظر بگیرید.

1-2 $S \to iEtSS' \mid a$

3-4 $S' \to eS \mid \varepsilon$

5 $E \to b$

جدول پارس گرامر فوق به‌صورت زیر است:

	i	t	a	e	b	$
S	1		2			
S'				3,4		4
E					5	

این گرامر LL(1) نیست، زیرا در خانهٔ M[S',e] جدول تجزیه آن دو شماره قاعده قرار دارد.

برای پی بردن به LL(1) بودن یک گرامر لازم نیست که حتماً جدول تجزیه آن به‌دست آید. با بررسی شرایط زیر نیز می‌توان LL(1) بودن یک گرامر را بررسی نمود. به عبارت دیگر گرامری LL(1) است که شرایط زیر در مورد قواعد به‌صورت $A \to \alpha | \beta$ آن صدق کند.

۱ـ $First(\beta) \cap First(\alpha) = \phi$

۲ـ حداکثر یکی از رشته‌های α و β رشته ε را تولید کنند.

۳ـ اگر $\alpha \Rightarrow^* \varepsilon$ در آن‌صورت $First(\beta) \cap Follow(A) = \phi$

به عنوان مثال، گرامر زیر را در نظر بگیرید.

1 $E \to TE'$

2 – 3 $E' \to +TE' | \varepsilon$

4 $T \to FT'$

5 – 6 $T' \to *FT' | \varepsilon$

7 – 8 $F \to (E) | id$

شرایط LL(1) بودن را برای گرامر فوق چک می‌کنیم. از آن‌جا که غیر پایانه‌های E و T تنها یک قاعده دارند، نیازی به بررسی ندارند در مورد غیر پایانه E' داریم:

$First(+TE') = \{+\}$. $First(\varepsilon) = \{\varepsilon\}$. $\{\varepsilon\} \cap \{+\} = \phi$

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

و برای غیر پایانه T' داریم:

$First(*FT') = \{*\}, First(\varepsilon) = \{\varepsilon\}, \{*\} \cap \{\varepsilon\} = \phi$

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

حال شرایط را برای غیر پایانه F بررسی می‌کنیم.

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

پس گرامر LL(1) است.

حال به عنوان مثال دیگر گرامر زیر را در نظر بگیرید.

$S \to iEtSS' | a$

$S' \to eS | \varepsilon$

$E \to b$

با توجه به آن‌که ε عضو $First(S')$ است، اگر شرط سوم را در مورد غیر پایانهٔ S' چک کنیم، مشخص خواهد شد که این گرامر LL(1) نیست.

$First(eS) = \{e\}, Follow(S') = \{e, \$\}, \{e, \$\} \cap \{e\} \neq \phi$

گرامرهایی که چپ‌گردی داشته باشند، LL(1) نیستند. برخی از گرامرها را می‌توان با حذف چپ‌گردی و فاکتورگیری از چپ به گرامر LL(1) تبدیل کرد، ولی فاکتورگیری و حذف چپ‌گردی باعث از بین رفتن خوانایی گرامرها می‌شوند. در ضمن تولید کد نیز مشکل‌تر می‌شود.

روش‌های اصلاح خطای نحوی در روش تجزیه LL(1)

از مهم‌ترین روش‌های اصلاح خطای قابل استفاده در تجزیه LL(1)، عبارتند از روش Panic Mode و روش Phrase Level . به‌طور کلی در روش LL(1) زمانی یک خطای نحوی تشخیص داده می‌شود که یا پایانهٔ بالای انباره با توکن جاری تطبیق نکند، و یا با غیر پایانهٔ بالای انباره A و توکن جاری a، خانهٔ M[A,a] خالی باشد.

در روش Panic mode اگر پارسر با مراجعه به یک خانه خالی جدول تجزیه یک خطای نحوی بیابد، آن‌قدر از رشته ورودی حذف می‌کند تا به یکی از اعضای مجموعه‌ای موسوم به مجموعه Synchronizing برسد. در روش Panic Mode به ازای هر غیر پایانه در گرامر یک مجموعه Synchronizing در نظر گرفته می‌شود. کارآیی روش Panic mode نیز بستگی به انتخاب مناسب مجموعه Synchronizing دارد. این مجموعه باید به گونه‌ای تعیین شود که عمل تجزیه بتواند بدون حذف قسمت زیادی از ورودی، به کار خود ادامه دهد.

یک انتخاب مناسب، در نظر گرفتن مجموعه Follow هر غیر پایانه‌ای به عنوان مجموعهٔ Synchronizing آن غیر پایانه است. با این وجود، در نظر گرفتن مجموعه Follow تنها برای Synchronizing کافی نیست. برای این که حذف کمتری در برنامه ورودی صورت بگیرد، می‌توان نمادهای بیش‌تری را به این مجموعه افزود. مثلاً می‌توان مجموعه First غیر پایانه‌ها را نیز به مجموعهٔ Synchronizing آن‌ها افزود. به عنوان یک مثال، از نحوه عمل پیاده‌سازی روش Panic Mode در تجزیه LL(1) گرامر زیر را در نظر بگیرید:

1 $E \rightarrow TE'$
2-3 $E' \rightarrow +TE' | \varepsilon$
4 $T' \rightarrow *FT' | \varepsilon$
7-8 $F \rightarrow (E) | id$

مجموعه Follow غیر پایانه‌ها را به عنوان Synchronizing آن‌ها در نظر گرفته و در جدول تجزیه در مقابل Follow غیر پایانه‌ها با گذاردن علامتی مثل "S" مجموعه Synchronizing هر غیر پایانه را معین می‌کنیم. به این ترتیب جدول پارس گرامر فوق، به صورت زیر در خواهد آمد.

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

برای اصلاح خطا به روش Panic Mode در الگوریتم تجزیه LL(1) به‌صورت صفحه بعد عمل می‌کنیم:

۱ـ اگر پارسر خانه M[A,a] را خالی ببیند، علامت a را در ورودی نادیده می‌گیرد.

۲ـ اگر در محل خانهٔ M[A,a] علامت "S" باشد غیر پایانهٔ بالای انباره حذف می‌شود. مشروط بر آن‌که A تنها غیر پایانه موجود در انباره نباشد.

۳ـ اگر پایانهٔ انباره با ورودی جاری تطبیق نکند، پایانهٔ بالای انباره حذف می‌شود.

(با استفاده از گرامر و جدول پارس مثال قبل) نشان می‌دهد: $id*+id$ شکل زیر مراحل تجزیه و اصلاح خطا را برای جملهٔ

محتوای انباره	ورودی	قاعده استفاده شده
$E	)id*+id$	Error → delete")"
$E	id*+id$	E → TE'
$E'T	id*+id$	T → FT'
$E'T'F	id*+id$	F → id
$E'T'id	id*+id$	
$E'T'	*+id$	T' → *FT'
$E'T'F*	*+id$	
$E'T'F	+id$	Error → delete"F"
$E'T'	+id$	T' → ε
$E'	+id$	E' → +TE'
$E'T+	+id$	
$E'T	id$	T → FT'
$E'T'F	id$	F → id
$E'T'id	id$	
$E'T'	$	T' → ε
$E'	$	E' → ε
$	$	End

در روش اصلاح خطای Phrase level در هر یک از خانه‌های خالی جدول تجزیه، یک نشانه‌رو، به زیر رویه‌ای جهت اصلاح خطا قرار داده می‌شود. این زیر رویه، عملیاتی نظیر تغییر، درج، و یا حذف را در ورودی و یا انباره انجام می‌دهند و هم‌چنین پیام‌های خطای مناسبی را نیز صادر می‌کنند.

تجزیه پایین به بالا (Bottom – Up Prasing)

یک روش کلی تجزیه پایین به بالا، روش انتقال ـ کاهش (Shift – Reduce) است. در این روش، عکس تجزیه بالا به پایین عمل می‌شود، به این ترتیب که از رشته ورودی شروع کرده و ساخت درخت تجزیه از برگ‌ها آغاز گشته و به طرف ریشه (علامت شروع) پیش می‌رود.

ترتیب به‌کارگیری قواعد در پارس بالا به پایین درست مطابق بسط چپ است. در حالی‌که ترتیب به‌کارگیری قواعد، در اکثر روش‌های تجزیه پایین به بالا درست عکس بسط راست است. گرامر زیر را در نظر بگیرید.

1 S → a A B e

2-3 A → A b c | b

4 B → d

جملهٔ abbcde را مورد بررسی قرار می‌دهیم. بسط راست این جمله به صورت زیر است:

$$S \xrightarrow[rm]{1} a\ ABe \xrightarrow[rm]{4} a\ Ade \xrightarrow[rm]{2} a\ Abcde \xrightarrow[rm]{3} abbcde$$

که در آن ترتیب به کار گرفتن قواعد گرامر، به صورت 1,4,2,3 (از چپ به راست) است. تجزیه پایین به بالای رشته فوق در جدول زیر آمده است:

مرحلهٔ تجزیه	فرم جمله‌ای تحت تجزیه	شماره قاعده	دست‌گیره
	S		
4	a ABe	1	a ABe
3	aAde	4	d
2	aAbcde	2	Abc
1	abbcde	3	b

به این ترتیب، جملهٔ abbcde به علامت شروع گرامر کاهش می‌یابد. ترتیب عملیات در این کاهش، درست برعکس بسط راست صورت گرفته است. در هر مرحله از کاهش، در پارس پایین به بالا این مشکل وجود دارد که پارسر، کدام زیر رشته را به عنوان دست‌گیره انتخاب و سپس از کدام قاعده برای کاهش آن استفاده نماید. در ادامه به ارایه چند تعریف در ارتباط با تجزیه پایین به بالا می‌پردازیم:

عبارت (Phrase): بخشی از یک فرم جمله‌ای است که از یک غیرپایانه به‌وجود آمده باشد. به عنوان نمونه در بسط زیر، β یک عبارت محسوب می‌شود.

$$S \Rightarrow^* \alpha A \gamma \Rightarrow^+ \alpha\beta\gamma$$

عبارت ساده (Simple Phrase): عبارتی است که در یک قدم به‌وجود آمده باشد. در بسط زیر β یک عبارت ساده است.

$$S \Rightarrow^* \alpha A \gamma \Rightarrow \alpha\beta\gamma$$

دست‌گیره (Handle): دست‌گیره عبارت ساده‌ای است که در جهت عکس یک بسط راست تولید شده باشد. در مثال زیر β یک دست‌گیره است. توجه داشته باشید که از آن‌جایی‌که دست‌گیره در رابطه با بسط راست مطرح است، سمت راست دست‌گیره هیچ غیرپایانه‌ای نیست. به همین خاطر در مثال زیر از "x" برای نمایش زیر رشته سمت راست دست‌گیره استفاده شده است.

$$S \underset{rm}{\Rightarrow}^* \alpha A x \underset{rm}{\Rightarrow} \alpha\beta x$$

اگر گرامر مورد استفاده گنگ نباشد، در هر مرحله از تجزیه پایین به بالا، تنها یک دست‌گیره وجود دارد. لیکن در صورت استفاده از یک گرامر گنگ، ممکن است در بعضی از قدم‌ها بیش‌تر از یک دست‌گیره موجود باشد. به مثال زیر توجه کنید:

1-4 $E \to E+E \mid E*E \mid (E) \mid id$

از آن‌جا که گرامر فوق، گنگ است، برای جملهٔ $id+id*id$ دو بسط راست و در نتیجه دو مسیر تجزیه پایین به بالا وجود دارد. این دو بسط در ادامه نشان داده می‌شود. همان‌گونه که مشاهده می‌شود، در قدم سوم تجزیه دو انتخاب برای دست‌گیره وجود دارد.

$E \underset{rm}{\Rightarrow} \underline{E+E}$ $\quad$ $E \underset{rm}{\Rightarrow} \underline{E*E}$

$\underset{rm}{\Rightarrow} E+\underline{E*E}$ $\quad$ $\underset{rm}{\Rightarrow} E*\underline{id}$

$\underset{rm}{\Rightarrow} E+E*\underline{id}$ $\quad$ $\underset{rm}{\Rightarrow} \underline{E+E*}\ id$

$\underset{rm}{\Rightarrow} E+\underline{id}*id$ $\quad$ $\underset{rm}{\Rightarrow} E+\underline{id}*id$

$\underset{rm}{\Rightarrow} \underline{id}+id*id$ $\quad$ $\underset{rm}{\Rightarrow} \underline{id}+id*id$

پیاده‌سازی روش تجزیه انتقال ـ کاهش با استفاده از یک انباره

در این روش، از یک انباره و یک میان‌گیر ورودی جهت نگهداری رشته‌ای که باید تجزیه شود، استفاده می‌گردد. در وضعیت شروع تجزیه به انتهای ورودی یک علامت "$" اضافه می‌گردد که خاتمه رشته ورودی، برای پارسر مشخص گردد. درون انباره نیز یک علامت "$" وارد می‌گردد.

پارسر آن‌قدر دو عمل انتقال و یا کاهش را انجام می‌دهد که یا یک خطای نحوی مشاهده گردد و یا این‌که به وضعیت خاتمهٔ پارس برسد. وضعیت خاتمه تجزیه، به این صورت است که توکن جاری علامت "$" است و درون انباره نیز تنها علامت شروع گرامر بر روی علامت "$" که در ابتدای تجزیه وارد انباره شده است، قرار دارد.

به‌طور رسمی‌تر، اعمالی که یک پارسر انتقال ـ کاهش انجام می‌دهد، عبارتند از:

۱ـ انتقال (Shift) : توکن جاری به بالای انباره انتقال می‌یابد. عمل انتقال تا زمانی ادامه می‌یابد که یک دست‌گیره در بالای انباره تشخیص داده شود.

۲ـ کاهش (Reduce): دست‌گیره‌ای در بالای انباره ظاهر شده است. دست‌گیره از بالای انباره حذف و به‌جای آن غیرپایانه سمت چپ قاعده‌ای که سمت راست آن مطابق دست‌گیره است، وارد انباره می‌شود.

۳ـ قبول ورودی (Accept) :پارسر پایان موفقیت‌آمیز تجزیه را اعلام می‌کند.

۴ـ تشخیص خطا (Error) : پارسر یک خطای نحوی، تشخیص داده و رویه خطاپرداز را فرا می‌خواند.

به عنوان نمونه، تجزیهٔ رشتهٔ id+id*id به روش انتقال ـ کاهش به‌صورت زیر است:

عمل انجام شده	باقی‌مانده ورودی	محتوای انباره
انتقال id	$ id+id*id $	$
کاهش بوسیلهٔ $E \rightarrow id$	$ +id*id $	$ id
انتقال +	$ +id*id $	$ E
انتقال id	$ id*id $	$ E+
کاهش بوسیلهٔ $E \rightarrow id$	$ *id $	$ E+id
انتقال *	$ id $	$ E+E
انتقال id	$ id $	$E+E*
کاهش بوسیلهٔ $E \rightarrow id$	$	$E+E*id
کاهش بوسیلهٔ $E \rightarrow E*E$	$	$E+E*E
کاهش بوسیلهٔ $E \rightarrow E+E$	$	$ E+E
Accept	$	$ E

در تجزیه به روش انتقال ـ کاهش مشکلات زیر وجود دارد:

۱ـ در تصمیم‌گیری در مورد این‌که کدام زیر رشته، تشکیل یک دست‌گیره می‌دهد.

۲ـ انتخاب قاعده‌ای که باید برای کاهش استفاده شود. این مشکل، زمانی بروز می‌کند که در سمت راست، بیش از یک قاعده با دست‌گیره مطابقت می‌کند. به چنین وضعیتی، تداخل کاهش ـ کاهش (Reduce/Reduce Conflict) گفته می‌شود.

انواع تداخل در تجزیه انتقال ـ کاهش

در پارس انتقال کاهش دو نوع تداخل می‌تواند روی دهد:

۱ـ تداخل انتقال ـ کاهش (Shift / Reduce Conflict) : زمانی روی می‌دهد که پارسر نتواند تصمیم بگیرد که عمل انتقال را باید انجام دهد یا عمل کاهش.

۲ـ تداخل کاهش ـ کاهش (Reduce / Reduce Conflict) : اگر بیش از یک قاعده جهت کاهش موجود باشد، این گونه تداخل روی خواهد داد.

به عنوان مثالی از تداخل نوع اول، گرامر زیر را در نظر بگیرید.

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

فرض کنید، توکن جاری "else" و رشته "if expr then stmt" بالای انباره قرار داشته باشد.

با توجه به وضعیت انباره، پارسر هم می‌تواند با استفاده از قاعدهٔ اول، عمل کاهش را انجام دهد و هم می‌تواند ابتدا توکن جاری را به بالای انباره انتقال داده و در زمان مناسب با استفاده از قاعدهٔ دوم، عمل کاهش را انجام دهد. حال به مثالی از تداخل نوع دوم توجه کنید. گرامر زیر را در نظر بگیرید:

1-2 stmt → id(parameter-list) | expr := expr

3-4 parameter-list → parameter-list , parameter | parameter

5 parameter → id

6-7 expr → id (expr-list) | id

8-9 expr-list → expr-list, expr | expr

قاعده شماره ۱ این گرامر، جهت توصیف فراخوانی رویه‌ها و قاعده ۶ گرامر، جهت توصیف مراجعه به آرایه‌ها است. فرض کنید به عنوان بخشی از ورودی، رشته "A(I,J)" آمده باشد. این بخش از ورودی به وسیله اسکنر به صورت "id(id,id)" تبدیل می‌گردد. هم‌چنین فرض کنید در وضعیتی از تجزیه قرار داریم که باقی‌مانده ورودی به‌صورت "$ ....id)," و زیر رشته "id (id" بالای انباره ظاهر شده باشد. در این حالت از دو قاعده، جهت عمل کاهش "id" می‌توان استفاده نمود (قواعد ۵ و ۷). یعنی یک تداخل کاهش / کاهش رخ داده است. در این مثال، انتخاب قاعده درست بستگی به نوع متغیر A دارد. در صورتی که A یک رویه باشد، باید از قاعده ۵ و اگر A یک آرایه باشد، باید از قاعدهٔ ۷ برای کاهش استفاده شود. یعنی پارسر باید با مراجعه به جدول نشانه‌ها و پی بردن به نوع A این تداخل را حل کند. راه‌حل دیگری نیز جهت رفع این نوع مشکل وجود دارد. اگر اسکنر در هنگامی که متغیر ورودی یک رویه است، به‌جای توکن "id" توکن دیگری، مثلاً "procid" به پارسر انتقال دهد، در این‌صورت، در موقع برخورد با وضعیت تداخل، کافیست پارسر داخل انباره و توکن زیر "(" را بررسی کند. اگر این توکن procid باشد، پارسر از قاعده شماره ۵ و در غیر این‌صورت از قاعده شماره ۷ جهت کاهش id بالای انباره استفاده می‌کند. توجه داشته باشید که در این صورت، قاعده شماره ۱ بایستی به‌صورت زیر اصلاح گردد.

1 stmt → procid (parameter-list)

روش تجزیه تقدم ـ عملگر (Operator - Precedence Parsing)

یکی از شرایطی که باید وجود داشته باشد تا بتوان از روش تجزیه تقدم عملگر استفاده نمود، این است که گرامر باید یک گرامر عملگر باشد.

گرامر عملگر ـ گرامری است که دارای خصوصیات زیر باشد:

۱ـ قاعده ε نداشته باشد.

۲ـ سمت راست هیچ قاعده‌ای دارای دو غیرپایانه مجاور نباشد.

به عنوان مثال، گرامر زیر گرامر عملگر نیست، زیرا سمت راست قاعدهٔ $E \to EAE$ دو غیر پایانهٔ مجاور دارد.

$$E \to EAE \mid (E) \mid -E \mid id$$

$$A \to + \mid - \mid * \mid / \mid \uparrow$$

اگر این گرامر را به صورت زیر تبدیل می‌کنیم، گرامر عملگر خواهد شد.

$$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid id$$

معایب روش پارس تقدم ـ عملگر: روش تقدم عملگر علیرغم داشتن مزیت پیاده‌سازی آسان، دارای معایبی نیز است. این معایب عبارتند از:

۱ـ به دلیل محدودیت‌هایی که دارد، گرامرهای کمی وجود دارند که بتوان از این روش برای آن‌ها استفاده کرد.

۲ـ در مورد اپراتورهایی مانند '-' (minus) که دارای دو تقدم متفاوتند (بسته به این که منهای unary است یا binary)، این روش کار نمی‌کند.

۳ـ روش چندان دقیقی نیست؛ یعنی ممکن است برخی از خطاهای نحوی را نتواند کشف کند.

در پارس، تقدم عملگر از سه رابطه تقدم مابین عملیات جهت هدایت عمل تجزیه، استفاده می‌گردد. در این روش روابط تقدم تنها بین پایانه‌های گرامر و به‌صورت زیر تعریف می‌گردد:

۱ـ $a<b$ یعنی پایانه a از پایانه b تقدم کمتری دارد. مانند $*<+$

۲ـ $a=b$ یعنی پایانه a و پایانهٔ b از تقدم یکسانی برخوردارند. (=)

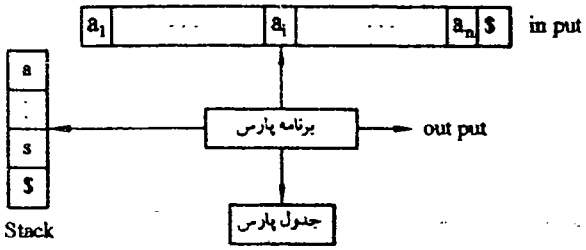
۳ـ $a>b$ یعنی پایانه a از پایانهٔ b تقدم بیش‌تری دارد. مانند $*<id$

روابط تقدمی که در این‌جا بین پایانه‌ها توصیف می‌شود، با روابط $>$،$=$ و $<$ معمولی که در بین اعداد طبیعی برقرار است، تفاوت زیادی دارند. به‌عنوان نمونه، در این‌جا با دانستن این‌که رابطهٔ $a<b$ برقرار است، نمی‌توان نتیجه گرفت که $b>a$. از طرفی ممکن است دو پایانه، هیچ‌یک از این سه رابطه تقدم را با هم نداشته باشند و یا این‌که دو پایانه دو رابطه تقدم متفاوت داشته باشند؛ مثلاً داشته باشیم که $*>-$ و هم $*<-$.

الگوریتم تجزیه تقدم ـ عملگر

ساختار پارسر در روش تقدم عملگر، مشابه ساختار یک پارسر LL(1) است. در این ساختار که در شکل زیر نشان داده شده است. مؤلفه اصلی پارسر، یک برنامه است که از یک طرف، ورودی خود را از اسکنر دریافت می‌کند و از یک انباره برای ذخیره اطلاعات و از یک جدول تجزیه برای هدایت عمل تجزیه استفاده می‌کند . در این روش، پارسر ابتدا یک علامت $ به انتهای رشتهٔ ورودی اضافه می‌کند. انباره نیز در ابتدای کار فقط شامل یک علامت $ است.

جدول پارسر در این روش، یک جدول دوبعدی مربع است که به تعداد پایانه‌های به‌علاوه یک (به خاطر علامت $) سطر و ستون دارد. در داخل جدول نیز در برخی خانه‌های جدول یکی از علامت‌های >، < و یا = و مابقی خانه‌های جدول خالی است.



با توجه به تعاریف توابع فوق رابطه‌های تقدم به صورت زیر تعریف می‌شوند:

$a = b$ iff $\exists U \rightarrow ...ab...$ or $U \rightarrow ...a\,W b...$

$a < b$ iff $\exists U \rightarrow ...aW...$ and $b \in \text{Firstterm}(W)$

$a > b$ iff $\exists U \rightarrow ...Wb...$ and $a \in \text{Lastterm}(W)$

حال به عنوان مثال، گرامر زیر را در نظر بگیرید:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

اگر توابع Firstterm و Lastterm را که تعریف رسمی آن‌ها در ادامه خواهد آمد، بر روی غیرپایانه‌های این گرامر اعمال کنیم، حاصل به‌صورت زیر خواهد شد:

$\text{Firstterm}(E) = \{+,*,(,id\}, \text{Firstterm}(T) = \{*,(,id\}, \text{Firstterm}(F) = \{(,id\}$

$\text{Lastterm}(E) = \{+,*,),id\}, \text{Lastterm}(T) = \{*,),id\}, \text{Lastterm}(F) = \{),id\}$

برای به‌دست آوردن رابطه علامت $ و سایر پایانه‌ها، قاعده‌ای به فرم $N \rightarrow \$S\$$ که در آن N یک غیرپایانه جدید و S علامت شروع گرامر است به گرامر اضافه می‌کنیم.

حال، با توجه به تعاریف روابط تقدم عملگر که در بالا عنوان شد، رابطه تساوی به سادگی با بررسی قواعد به‌دست می‌آید. در گرامر فوق بادر نظر گرفتن قاعده جدیدی که به خاطر علامت $ اضافه می‌گردد، دو رابطه تساوی وجود دارد: $=$ و (=). توجه داشته باشید که (=) هیچ اطلاعی در مورد "("و")" به ما نمی‌دهد؛ مثلاً از این نمی‌توان نتیجه گرفت که)=(.

برای یافتن روابط >، با توجه به تعریف آن به دنبال نقاطی در گرامر می‌گردیم که یک پایانه در سمت چپ یک غیرپایانه قرار گرفته باشد. مثلاً در قاعده پنجم این گرامر، ")" سمت چپ غیرپایانه E قرار گرفته است. حال اگر پایانه‌ای مثلاً b عضو مجموعه

Firstterm(E) باشد، رابطه تقدم a<b) برقرار است. در واقع این رابطه بین "(" و همه اعضا Firstterm(E) برقرار است. یعنی، در این‌جا می‌توان نتیجه گرفت که روابط (+>)،(*>)،)>) و (id>) برقرار است. به همین ترتیب، می‌توان سایر روابط تقدم > را نیز یافت.

برای یافتن روابط < با توجه به تعریف آن به دنبال نقاطی در گرامر می‌گردیم که یک پایانه در سمت راست یک غیرپایانه قرار گرفته باشد. مثلاً در قاعده پنجم این گرامر "(" سمت راست غیرپایانه E قرار گرفته است. حال اگر پایانه‌ای مثلاً a عضو مجموعه Lastterm(E) باشد، رابطه تقدم (a< برقرار است. در واقع، این رابطه نیز بین " (" و همه اعضا Lastterm(E) برقرار است. یعنی در این‌جا، می‌توان نتیجه گرفت که روابط (<+،(<*،(<(و (id< برقرار است. به همین ترتیب، می‌توان سایر روابط تقدم < را نیز یافت.

در نهایت، جدول روابط تقدم گرامر فوق به‌صورت زیر خواهد بود. محل‌های خالی در جدول نشانگر آن است که دو پایانه با هم رابطه ندارند. مثلاً در این‌جا id با id رابطه ندارد. این بدان معنی است که با استفاده از گرامر فوق، نمی‌توان فرم جمله‌ای تولید نمود که در آن دو id مجاور هم قرار بگیرند. (در این‌جا منظور از مجاور بودن دو پایانه این است که یا دقیقاً مجاور باشند و یا این‌که بین آن‌ها یک غیرپایانه باشد. توجه داشته باشید که به دلیل محدودیت خاصی که روی قواعد گرامر عملگر وجود دارد، امکان ندارد که در یک فرم جمله‌ای، دو غیر پایانهٔ مجاور هم قرار بگیرند. بنابراین، حداکثر ممکن است بین دو پایانه، یک غیرپایانه قرار داشته باشد که در این صورت نیز طبق تعریف آن دو پایانه مجاور هم محسوب می‌شوند.)

	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
$	<	<	<		<	=

در این روش، برنامهٔ پارس در هر قدم از پارس، با استفاده از بالاترین پایانهٔ انباره (a) و توکن جاری (b) (به غیر پایانهٔ بالای انباره توجهی ندارد) به عنوان اندیس جدول تجزیه و مراجعه به این جدول، یکی از اعمال زیر را انجام می‌دهد:

۱ـ اگر رابطهٔ پایانهٔ بالای انباره توکن جاری به‌صورت a = b باشد، پارسر فقط توکن جاری را به بالای انباره انتقال می‌دهد.

۲ـ اگر رابطهٔ پایانهٔ انباره و توکن جاری به‌صورت a < b باشد، پارسر ابتدا علامت< و سپس توکن جاری را به بالای انباره انتقال می‌دهد.

۳ـ اگر رابطهٔ پایانهٔ بالای انباره و توکن جاری به صورت a>b باشد، پارسر عمل کاهش را انجام می‌دهد. برای این‌کار، در داخل انباره آن‌قدر پایین می‌رود تا به اولین علامت < برسد، دست‌گیره، رشته مابین این علامت و بالای انباره است (به‌علاوه غیرپایانهٔ زیر علامت < در صورت وجود). پارسر برای عمل کاهش، دست‌گیره پیدا شده را از انباره حذف و به‌جای آن یک غیرپایانه نوعی (مثلاً N) وارد انباره می‌کند. (در روش تقدم، عملگر پس از تهیه جدول تجزیه از روی گرامر، دیگر بین غیرپایانه‌های گرامر تمایزی قایل نمی‌شویم و به‌جای همه آن‌ها می‌توان از یک غیرپایانه نوعی استفاده نمود. همین عامل، ضعف این روش در کشف برخی از خطاهای نحوی گردیده است.)

۴ـ اگر پایانهٔ بالای انباره با ورودی جاری رابطه‌ای نداشته باشد، یک خطای نحوی است و پارسر رویهٔ اصلاح خطا را فرا می‌خواند.

حال، به عنوان یک مثال به تجزیه رشته id+id*id با استفاده از جدول تجزیه گرامر غیر گنگ عبارات جبری توجه کنید. جدول تجزیه و مراحل تجزیه به‌صورت قدم به قدم در شکل‌های زیر آمده است. در قدم‌هایی که عمل کاهش صورت گرفته، زیر دست‌گیره خط کشیده شده است.

عمل انجام شده	باقی‌مانده ورودی	محتوای انباره
انتقال < و id	id+id*id $	$
کاهش بوسیلهٔ E → id	+id *id $	$ ≤ id
انتقال > و +	+id *id $	$ E
انتقال < و id	id *id $	$ E<+
کاهش بوسیلهٔ E → id	*id $	$ E<+≤id
انتقال > و *	*id$	$ E<+E
انتقال < و id	id $	$E < +E < *
کاهش بوسیلهٔ E → id	$	$E<+E<*≤ id
کاهش بوسیلهٔ E → E*E		$E<+E<*E
کاهش بوسیلهٔ E → E+E		$E<+E
پایان پارس		$ E

نحوه یافتن روابط تقدم

برای تعیین روابط تقدم از دو تابع با تعریف زیر استفاده می‌کنیم. این دو تابع، روی غیرپایانه‌ها تعریف شده‌اند و حاصل آن‌ها مجموعه‌ای از پایانه‌ها است.

$$\text{Firstterm}(A)=\{a \mid A \Rightarrow^{+} a\alpha \quad \text{or} \quad A \Rightarrow^{+} Ba\alpha\}$$

$$\text{Lastterm}(A)=\{a \mid A \Rightarrow^{+} \alpha a \quad \text{or} \quad A \Rightarrow^{+} \alpha aB\}$$

که در آن a یک پایانه، B یک غیرپایانه و α یک رشته از پایانه و غیر پایانه است.

اصلاح خطا در روش تقدم عملگر

در این روش، کلاً در دو صورت یک خطای نحوی تشخیص داده می‌شود. اول، وقتی که هیچ رابطه‌ای بین پایانهٔ بالای انباره و ورودی جاری نباشد؛ و دوم، هنگامی که دست‌گیره‌ای بالای انباره با سمت راست هیچ قاعده‌ای تطبیق نکند.

برای اصلاح خطاهای نوع اول در خانه‌های خالی جدول، نشانه‌روهایی به زیرروال‌های اصلاح خطا می‌گذاریم، به‌طوری‌که اگر در عمل تجزیه به یک خانه خالی جدول رجوع شد، زیر روال مربوطه فراخوانی شده و خطا به نحو مقتضی اصلاح گردد.

به عنوان نمونه گرامر زیر را در نظر بگیرید:

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

جدول تجزیه این گرامر به‌صورت زیر است:

	id	(	)	$	+
$	<	<	e1	=	<
)	e2	e2	>	>	>
id	e2	e2	>	>	>
(	<	<	=	E3	<
+	<	<	>	>	>

روال‌های اصلاح به‌صورت زیر تعریف می‌شوند:

e1 : توکن "(" را حذف و پیغام "در ورودی یک پرانتز بسته اضافی وجود دارد " را چاپ کن.

e2 : پایانه "+" را به ورودی اضافه و پیغام " یک عملگر در برنامه کم است" را چاپ کن.

e3 : پایانه ")" را از بالا حذف و پیغام " یک پرانتز بسته در ورودی کم است را چاپ کن.

در صورت تشخیص خطای نوع دوم، یعنی عدم تطبیق دست‌گیره با سمت راست هیچ‌یک از قواعد گرامر، پارسر به دنبال قاعده‌ای که سمت راست آن شبیه دست‌گیره باشد (در یک یا دو علامت تفاوت داشته باشند) جستجو می‌کند و با توجه به اختلاف دست‌گیره و سمت راست قاعده پیدا شده، پیغام مناسبی چاپ می‌کند و عمل کاهش را انجام می‌دهد.

مثلاً، فرض کنید دستگیرهٔ aNbc باشد و قاعده‌ای به صورت (aEc → ...) پیدا شود. از آن‌جایی که غیرپایانه‌ها در این روش تجزیه، اهمیتی ندارند و تنها محل آن‌ها در انباره اهمیت دارد؛ لذا در مقایسه دست‌گیره با سمت راست قواعد، تنها به موقعیت غیرپایانه‌ها اهمیت داده می‌شود. در این مثال، با توجه به این‌که اختلاف دست‌گیره و سمت راست قاعده در پایانه "b" است، پیغام زیر صادر می‌شود. توجه داشته باشید که پایانه‌های اضافی در دست‌گیره نشانه علایم اضافی در برنامه ورودی است.

Illegal "b" on line...

حال، اگر دست‌گیره به‌صورت abEc باشد و سمت راست قاعده‌ای پیدا شده به‌صورت abEdc باشد، پیغام زیر صادر خواهد شد.

Missing "d" on line...

ممکن است اختلاف در مورد یک غیرپایانه باشد. به عنوان مثال، فرض کنید abc دست‌گیره و aEbc سمت راست قاعده‌ای از گرامر باشد. در این صورت صدور پیغامی به‌صورت "Missing E on line" مجاز نیست. زیرا کاربر یک کامپایلر، اطلاعی در مورد غیرپایانه‌های گرامر ندارد و لذا در چاپ نبایستی از غیر پایانه‌ها استفاده نمود. در این حالت بایستی با توجه به ساختار نحوی که غیرپایانه موردنظر توصیف می‌کند، درباره خطای کشف شده گزارش داد. مثلاً اگر E معرف یک عبارت جبری (در ساده‌ترین شکل یک عملوند) است، می‌توان پیغام زیر را صادر نمود:

Missing operand on line...

روش تجزیه تقدم ساده (Simple Precedence Parsing)

این روش تجزیه، بسیار شبیه روش تجزیه تقدم عملگر است و در واقع، بهبود یافته تقدم عملگر است. در این روش، روابط تقدم بین همهٔ عناصر گرامر تعریف شده، در حالی‌که در تقدم عملگر، این روابط فقط بین پایانه‌ها تعریف می‌شود. برای استفاده از این روش، محدودیت‌های کمتری نسبت به مورد تقدم عملگر وجود دارد که باعث می‌شود که روش تقدم ساده، طیف بیش‌تری از گرامرها را در بر بگیرد. به عنوان نمونه، در این‌جا وجود غیرپایانه‌های مجاور در سمت راست قواعد مجاز است. لیکن، مانند حالت قبل وجود قواعد اپسیلون مجاز نیست. از آن‌جا که در روش تقدم ساده، بر خلاف روش تقدم عملگر بین غیرپایانه‌ها تمایز قایل می‌شویم، در این‌جا یک محدودیت جدید داریم که سمت راست هیچ دو قاعده‌ای باید یکسان باشد. زیرا در غیر این‌صورت در بعضی از قدم‌ها تداخل کاهش -

کاهش پیش خواهد آمد. البته، این محدودیت چندان مهم نیست. در هر دو مورد این روش‌ها یک محدودیت مشترک وجود دارد که در خانه‌های جدول تجزیه بایستی حداکثر یک رابطه تقدم وجود داشته باشد.

در روش تقدم ساده هم برای هدایت عملیات از روابط سه گانه تقدم استفاده می‌شود. البته، در روش تقدم ساده این روابط بین کلیه علایم گرامر (پایانه و غیرپایانه و $) تعریف می‌شود. جدول تجزیه تقدم ساده یک جدول مربع است که به تعداد حاصل جمع تعداد پایانه‌ها و غیرپایانه‌های گرامر به‌علاوه یک (به خاطر علامت $) سطر و ستون دارد.

برای تعیین روابط تقدم ساده، از دو تابع با نام‌هایHead و Tail استفاده می‌شود که تعریف رسمی آن‌ها به‌صورت زیر است:

هر دو این توابع بر روی یک غیرپایانه عمل کرده و حاصل آن‌ها مجموعه‌ای از علایم گرامر است.

$$HEAD(U)=\{X|U\Rightarrow^{+} X\alpha\}$$

$$TAIL(U)=\{X|U\Rightarrow^{+} \alpha X\}$$

با استفاده از دو تابع فوق، روابط تقدم ساده به‌صورت زیر تعریف می‌شوند:

$X = Y$ iff $\exists U \rightarrow ...XY...$

$X < Y$ iff $\exists U \rightarrow ...XA...$ and $Y \in HEAD(A)$

$X > Y$ iff $\exists U \rightarrow ...AB...$ and $X \in TAIL(A)$ and $(Y \in HEAD(B)$ or $Y = B)$

به عنوان مثال گرامر زیر را در نظر بگیرید:

$S \rightarrow (S\ S)$

$S \rightarrow c$

مانند روش تقدم عملگر، ابتدا قاعده‌ای به‌فرم $N \rightarrow \$S\$$ به قواعد گرامر اضافه کرده، سپس مطابق روالی که در آن‌جا ذکر گردید، به دنبال قواعدی می‌گردیم که شرایط تعاریف فوق در مورد آن‌ها صدق نماید. حاصل این کار در مورد مثال فوق به‌صورت جدول تجزیه زیر خواهد بود. برای تفکیک روابط تقدم ساده از روابط تقدم عملگر، روابط تقدم ساده با استفاده از علایم متفاوتی نمایش داده خواهد شد:

	S	$	(	)	c
S	=	=	<	=	<
$	=		<		<
(	=		<		<
)		>	>	>	>
c		>	>	>	>

الگوریتم تجزیه به روش تقدم ساده

در این روش، پارسر در هر قدم از تجزیه با استفاده از توکن جاری b و عنصر بالای انباره X (که می‌تواند پایانه یا غیرپایانه باشد) به جدول تجزیه مراجعه کرده و به‌صورت یکی از حالات زیر عمل می‌کند:

۱ـ در صورتی که رابطه علامت بالای انباره و توکن جاری به‌صورت $X \lessdot b$ باشد، پارسر عمل انتقال را انجام می‌دهد. در این مورد ابتدا علامت $\lessdot$ و سپس توکن جاری b را به بالای انباره منتقل می‌کند.

۲ـ در صورتی که رابطه دو عنصر مزبور به‌صورت $X \doteq b$ باشد، پارسر فقط توکن جاری را به بالای انباره انتقال می‌دهد.

۳ـ در صورتی که رابطه به‌صورت b ⋗ X باشد، پارسر عمل کاهش را انجام می‌دهد. در این حالت، دست‌گیره رشته بالای انباره تا اولین علامت ⋖ است. پارسر ابتدا دست‌گیره را از بالای انباره حذف می‌کند. اگر عنصر بالای انباره (پس از حذف دست‌گیره) را Top بنامیم و سمت چپ قاعده‌ای را که پارسر از آن جهت کاهش استفاده می‌کند، Lhs بنامیم، پارسر رابطه بین Lhs و Top را از جدول استخراج نموده و یکی از اعمال زیر را انجام می‌دهد:

ـ اگر رابطه Top و Lhs به‌صورت Lhs ⋖ Top باشد، پارسر ابتدا علامت ⋖ و سپس Lhs را وارد انباره می‌کند.

ـ اگر رابطه Top و Lhs به‌صورت Lhs ⋗ Top باشد، پارسر فقط Lhs را وارد انباره می‌کند.

ـ اگر Top و Lhs رابطه‌ای نداشته باشند، یک خطای نحوی رخ داده است و بایستی رویه اصلاح خطا فراخوانده شود.

۴ـ در صورتی که عنصر بالای انباره X و ورودی جاری b رابطه‌ای نداشته باشند، یک خطای نحوی رخ داده است و بایستی رویه اصلاح خطا فراخوانده شود.

۵ـ در صورتی که توکن جاری $ و در داخل انباره $S (S علامت شروع گرامر است) باقی مانده باشد، پارسر، پایان موفقیت‌آمیز تجزیه را اعلام می‌کند.

مشکلات چپ‌گردی و راست‌گردی در روش تقدم ساده

هر گاه، در قواعد گرامر وضعیتی به صورت زیر باشد که در آن قاعده اول، یک قاعده چپ‌گرد است، بین علامت X و غیرپایانه U دو رابطه تقدم به‌صورت زیر وجود دارد:

$U \rightarrow U...$

$V \rightarrow ...XU...$

$X \doteq U, X \lessdot U$

برای حل این مشکل قواعد فوق را به‌صورت زیر تبدیل می‌کنیم که در آن W یک غیرپایانه جدید است.

$U \rightarrow U$

$V \rightarrow ...XW...$

$W \rightarrow U$

حال روابط تقدم بین این علایم به‌صورت زیر است:

$X \doteq W, X \lessdot U$

هم‌چنین هر گاه در قواعد گرامر وضعیتی به صورت زیر باشد، که در آن قاعده اول، یک قاعده راست‌گرد است، بین غیرپایانه U و علامت X دو رابطه تقدم وجود دارد:

$U \rightarrow ...U$

$V \rightarrow ...UX...$

برای رفع این مشکل قواعد را به‌صورت زیر تغییر می‌دهیم:

$U \rightarrow ...U$

$V \rightarrow ...WX...$

$W \rightarrow U$

در ادامه به عنوان یک نمونه از تجزیه به روش تقدم ساده، به تجزیه جمله ((c(cc) توجه کنید:

عمل انجام شده	باقی‌مانده ورودی	محتوای انباره
انتقال	(c(cc)) $	$
انتقال	c(cc)) $	$<(
کاهش به‌وسیله S→c	(cc)) $	$<(<c
انتقال	cc)) $	$<(S
انتقال	c)) $	$<(S<(
کاهش با S→c	c)) $	$<(S<(<c
انتقال	)) $	$<(S<(S
کاهش با S→c	)) $	$<(S<(S<c
انتقال	)) $	$<(S<(SS
کاهش بوسیلهٔ S→(SS)	)$	$<(S<(SS)
انتقال	)$	$<(SS
کاهش بوسیلهٔ S→(SS)	$	$<(SS)
پایان	$	$S

روش‌های تجزیه LR

یکی از قوی‌ترین روش‌های تجزیه پایین به بالا که می‌تواند در مورد اکثر گرامرهای مستقل از متن استفاده شود، روش LR است. مزایای روش‌های تجزیه LR عبارتند از:

۱ـ تقریباً تمامی ساختارهای زبان برنامه‌نویسی را می‌توان توسط پارسرهای LR تشخیص داد.

۲ـ روش تجزیه LR کلی‌ترین روش تجزیه بدون پی‌جویی و به طریقه انتقال ـ کاهش است که تاکنون شناخته شده و می‌توان آن را به کارآیی هر روش دیگری پیاده‌سازی کرد.

۳ـ مجموعه زبان‌هایی که توسط روش LR تجزیه می‌شوند یک ابر مجموعه کامل، از زبان‌هایی هستند که توسط پارسرهای پیش‌گو تجزیه می‌شوند.

۴ـ یک پارسر LR خطاهای نحوی را در کمترین زمان ممکن، توسط بررسی چپ به راست ورودی پیدا می‌کند.

تجزیه LR(1) خود از سه روش زیر تشکیل شده است:

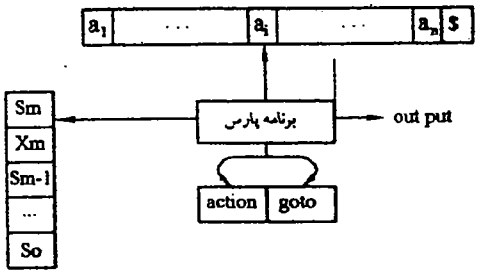
۱ـ SLR (Simple LR)

۲ـ LALR (LookAhead LR)

۳ـ CLR (Canonical LR)

شاید تنها عیب روش تجزیه LR، حجم کار بسیار زیادی است که در پیاده‌سازی دستی آن لازم است. به عبارت دیگر، پیاده‌سازی دستی روش‌های LR مشکل است؛ لیکن نرم‌افزارهایی وجود دارد (مثلاً YACC)که با کمک آن‌ها می‌توان پارسرهای LR را به‌طور خودکار تولید نمود.

ساختار کلی پارسرهای LR به فرم زیر است:



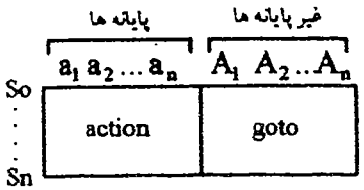
برنامه تجزیه هر سه روش فوق، مشابه یکدیگر است و تنها جدول تجزیه آن‌ها متفاوت است.

فرم کلی رشته‌هایی که در انباره این نوع پارسرها قرار می‌گیرد، به‌صورت زیر است:

$$S_0X_1S_1X_2S_2...X_mS_m$$

که در آن هر X_i یک علامت گرامر و هر S_i نشان‌گر یک وضعیت است. برای تهیه جدول تجزیه روش‌های LR بایستی دیاگرام انتقال ویژه‌ای رسم گردد که در بخش‌های بعد توضیح داده خواهد شد.

جدول تجزیه در روش LR از دو بخش action و goto تشکیل شده است. در خانه‌های غیر خالی بخش action دستوراتی قرار می‌گیرد (خانه‌های خالی نمایانگر خطای نحوی هستند) که عمل تجزیه را هدایت می‌کنند.



قبل از توضیح الگوریتم تجزیه LR لازم است با مفهومی بنام پیکربندی (Configuration) آشنا شویم. پیکربندی عبارت است از زوج مرتب:

$$\langle S_0X_1S_1X_2S_2...X_mS_m \quad , \quad a_ia_{i+1}...a_n\$ \rangle$$

که در آن $S_0X_1S_1X_2S_2...X_mS_m$ محتوای انباره و $a_ia_{i+1}...a_n\$$ باقی‌مانده ورودی است. یک پیکربندی، نشان دهنده وضعیت عمل تجزیه در یک لحظه خاص است.

الگوریتم تجزیه LR

پارسر LR در هر قدم از تجزیه، از شماره وضعیت بالای انباره (S_m) و توکن جاری (a_i) به‌عنوان اندیس جدول تجزیه استفاده کرده و به خانه action $[S_m, a_i]$ مراجعه می‌کند. در خانه مزبور، اگر خالی نباشد، یکی از سه دستور (Shift S)، (Reduce $A \to \alpha$) و یا (Accept) قرار دارد که پارسر بر اساس محتوای آن به‌صورت زیر عمل می‌کند:

۱ـ اگر action $[Sm,a_i]$=Shift S باشد، پارسر عمل انتقال را انجام داده و به پیکربندی زیر وارد می‌شود.

$(S_0X_1S_1X_2S_2...X_mS_ma_i\ S.a_{i+1}...a_n\ S)$

۲ـ اگر action$[S_m,a_i]$=Reduce $A \to \alpha$ باشد، پارسر عمل کاهش را انجام داده و به پیکربندی زیر وارد می‌شود،

که در آن $r=|\alpha|$ است (r برابر با طول سمت راست قاعده $A \to \alpha$ است) و goto$[S_{m-r}, A]$=S است.

$(S_0X_1S_1X_2S_2...X_{m-r}S_{m-r}\ A\ S.a_i\ a_{i+1}...a_n\ S)$

۳ـ اگر action$[S_m,a_i]$=Accept باشد، پارسر پایان موفقیت‌آمیز تجزیه را اعلام می‌دارد.

۴ـ اگر خالی =action$[S_m,a_i]$ باشد، پارسر یک خطای نحوی تشخیص داده و رویه اصلاح خطا را فرا می‌خواند.

به عنوان یک مثال، گرامر زیر و جدول تجزیه آن را در نظر بگیرید. برای صرفه‌جویی در فضا، در جداول تجزیه LR دستورات کاهش به‌صورت "rn" که در آن n شماره قاعده‌ای است که از طریق آن عمل کاهش انجام می‌گردد، نشان داده می‌شود. هم‌چنین دستورات انتقال نیز به‌صورت "S_n" که در آن n شماره یک وضعیت است، نشان داده می‌شود.

1 – 2 $E \to E+T \mid T$

3 – 4 $T \to T*F \mid F$

5 – 6 $F \to (E) \mid id$

	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				acc			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

مراحل تجزیه عبارت id*id+id به‌صورت زیر خواهد بود:

عمل انجام شده	باقی‌مانده ورودی	محتوای انباره
انتقال S_5	id*id+id $	0
کاهش بوسیلهٔ $F \rightarrow id$	*id +id $	0 id 5
کاهش بوسیلهٔ $T \rightarrow F$	*id +id $	0 F 3
انتقال	*id +id $	0 T2
انتقال	id +id$	0 T 2 *7
کاهش بوسیلهٔ $F \rightarrow id$	+id$	0 T 2 *7id 5
کاهش به‌وسیله $T \rightarrow T*F$	+id $	0 T 2 * 7 F10
کاهش به‌وسیله $E \rightarrow T$	+id $	0 T 2
انتقال	+id $	0 E 1
انتقال	+id $	0 E 1 + 6
کاهش بوسیلهٔ $F \rightarrow id$	$	0 E 1 + 6 id 5
کاهش به‌وسیله $T \rightarrow F$	$	0 E 1 +6F3
کاهش به‌وسیله $E \rightarrow E+T$	$	0 E 1+6T9
Accept	$	0 E 1

نحوه تهیه جدول تجزیه SLR(1)

در میان سه روش LR ، ساده‌ترین روش از نظر پیاده‌سازی روش SLR و یا Simple LR است. قبل از توضیح نحوه به‌دست آوردن جدول تجزیه SLR(1)، لازم است با چند مفهوم جدید آشنا شویم.

آیتم LR(0) : یک آیتم LR(0)، یک قاعده از گرامر است که در سمت راست آن در محلی یک علامت خاص (مثلاً •) قرار گرفته است. مثلاً اگر قاعده‌ای به‌صورت $A \rightarrow XYZ$ داشته باشیم، می‌توان از روی آن چهار آیتم زیر را ایجاد کرد.

$A \rightarrow \bullet XYZ$
$A \rightarrow X \bullet YZ$
$A \rightarrow XY \bullet Z$
$A \rightarrow XYZ \bullet$

از روی قواعد اپسیلون $A \rightarrow \varepsilon$ تنها یک آیتم به فرم $A \rightarrow \bullet$ می‌توان ایجاد نمود.

تابع بستار (Closure) : اگر I یک مجموعه از آیتم‌های یک گرامر باشد، closure(I) نیز یک مجموعه از آیتم‌ها است که به صورت زیر محاسبه می‌شود.

۱ـ ابتدا هر آیتم که در I وجود دارد را به closure (I) اضافه می‌کنیم.

۲ـ سپس اگر قاعده‌ای به فرم $A \rightarrow \alpha \bullet B\beta$ در closure(I) باشد و قاعده‌ای به فرم $B \rightarrow \gamma$ داشته باشیم، $B \rightarrow \bullet\gamma$ را نیز به closure (I) اضافه می‌کنیم. این قدم را آن‌قدر ادامه می‌دهیم، تا دیگر چیزی به closure (I) اضافه نشود.

رسم دیاگرام انتقال SLR

همان‌گونه که قبلاً اشاره شد، برای تهیه جدول تجزیه روش‌های LR بایستی یک دیاگرام انتقال رسم گردد. این امر در مورد روش SLR(1) به‌صورت زیر اجرا می‌گردد:

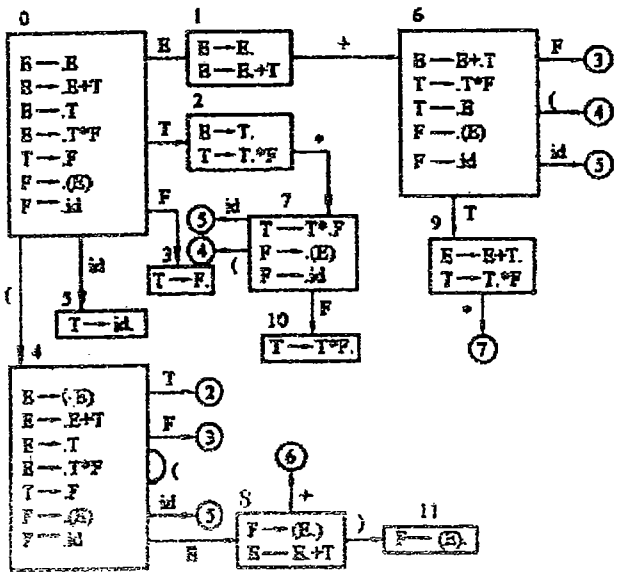
ابتدا قاعده‌ای به فرم $S' \rightarrow S$ که در آن S علامت شروع گرامر و S' یک غیرپایانه جدید است، به گرامر اضافه می‌کنیم. گرامر حاصل را گرامر افزوده (Augmented Grammar) گویند.

رسم دیاگرام را با وضعیت S_0 و آیتم $S' \rightarrow \bullet S$ شروع می‌کنیم و سپس بستار این آیتم را محاسبه کرده و در S_0 قرار می‌دهیم. سپس در صورتی‌که در حالت S_0 (یا به‌طور کلی S_i) آیتم‌هایی به‌صورت زیر (که در آن x می‌تواند یک پایانه و یا غیرپایانه باشد) وجود داشته باشد،

$$\begin{bmatrix} A_1 \rightarrow \alpha_1 \bullet x\beta \\ \vdots \\ An \rightarrow \alpha_n \bullet x\beta_n \end{bmatrix}$$

وضعیت جدیدی به نام S_j ایجاد کرده، S_i را توسط لبه‌ای با برچسب x به S_j متصل می‌کنیم و آیتم‌های فوق را با این تغییر که در همه، علامت • به بعد از علامت x منتقل شده است، در وضعیت جدید قرار می‌دهیم. سپس بستار این آیتم‌ها را محاسبه و در S_j قرار می‌دهیم. چنان‌چه در دیاگرام حالتی مانند S_k وجود داشته باشد که دقیقاً مطابق S_j باشد، حالت S_j ایجاد نشده و در عوض S_i توسط لبه‌ای با برچسب x به S_k متصل می‌گردد. این قدم را آن‌قدر تکرار می‌کنیم، تا دیگر حالت جدیدی به دیاگرام اضافه نگردد.

به عنوان نمونه، دیاگرام SLR گرامر عبارات جبری که در بالا آمده است، به صورت زیر است:



تهیه جدول تجزیه از روی دیاگرام SLR(1)

پس از رسم دیاگرام SLR(1) یک گرامر، جدول تجزیه آن‌را به صورت زیر تکمیل می‌کنیم. ابتدا نحوه تکمیل بخش action جدول توضیح داده می‌شود.

۱ـ اگر با ورودی a از حالت i به حالت j می‌رویم، در خانه action [i,a] جدول دستور Shift j را قرار می‌دهیم.

۲ـ اگر در حالت i آیتمی به فرم $A \to \alpha \bullet$ داریم، به ازای هر b متعلق به مجموعه Follow(A) در خانه‌های action[i,b] دستور Reduce n را قرار می‌دهیم. که در آن n شماره قاعده $A \to \alpha$ است.

۳ـ اگر در حالت i آیتم $S' \to S \bullet$ قرار داشته باشد، در خانه action[i,$] دستور accept را قرار می‌دهیم.

۴ـ در خانه‌های خالی بخش action می‌توان علامتی به عنوان خطا قرار داد.

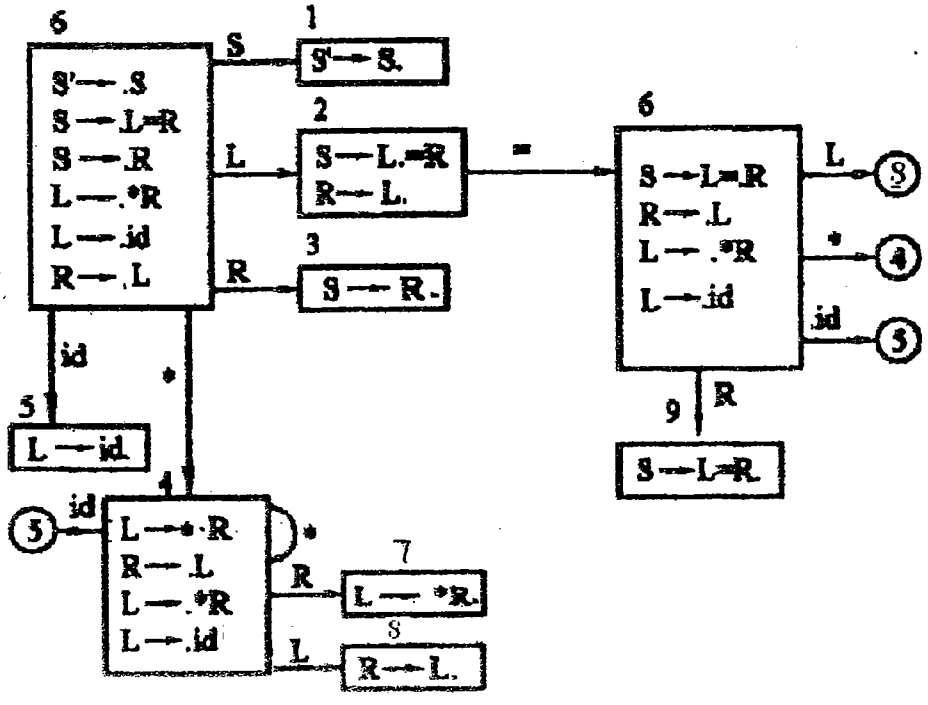
پس از تکمیل بخش action جدول تجزیه، محتوای خانه‌های بخش goto جدول به این صورت تعیین می‌گردد: اگر با غیر پایانه A از حالت i به حالت j می‌رویم، در خانه goto[i,A] مقدار j را قرار می‌دهیم.

در صورتی که دیاگرام و جدول تجزیه SLR(1) گرامری به‌دست آوریم و در خانه‌های بخش action جدول حداکثر یکی ازدستورات shift ، accept و یا reduce قرار بگیرد، آن گرامر SLR(1) است. (اگر گرامری SLR(1) باشد، حتماً CLR(1) ، LALR(1) نیز است. این رابطه در مورد حالت کلی LR(k) نیز برقرار است. یعنی در حالت کلی اگر گرامری SLR(k) باشد، حتماً LALR(k) و CLR(k) نیز است.) حال اگر در حداقل یکی از خانه‌های مذکور بیش از یک دستور قرار بگیرد، گرامر مربوطه SLR(1) نیست. به گرامر زیر توجه کنید. دیاگرام و جدول تجزیه این گرامر در شکل زیر رسم شده است.

1−2 $S \to L = R \mid R$

3−4 $L \to *R \mid id$

5 $R \to L$



	Action				Goto		
	=	*	id	$	S	R	L
0		S_4	S_5		1	3	2
1				Acc			
2	S6/r5			r_5			
3				r_2			
4		S_4	S_5			7	8
5	r_4			r_4			
6		S_4	S_5			9	8
7	r_3			r_3			
8	r_5			r_5			
9				r_1			

چون در خانه action[2,=] یک تداخل انتقال/کاهش وجود دارد، گرامر فوق SLR(1) نیست. لیکن هنوز ممکن است مثلاً SLR(2) و یا LALR(1) باشد.

آیتم‌های هسته‌ای (kernel items): در یک دیاگرام SLR به کلیه آیتم‌هایی که علامت • در ابتدای سمت راست آن‌ها قرار نگرفته و به آیتم خاص $S' \rightarrow \bullet S$، آیتم‌های هسته‌ای گویند. به آیتم‌هایی که علامت مزبور در ابتدای سمت راست آن‌ها قرار دارد، (به غیر از $S' \rightarrow \bullet S$) آیتم غیرهسته‌ای گویند. آیتم‌های هسته‌ای دارای اهمیتی بیش‌تر می‌باشند، زیرا در صورت کمبود حافظه در تهیه دیاگرام SLR یک گرامر، می‌توان به ذخیره صرفاً آیتم‌های هسته‌ای اکتفا نمود. سایر آیتم‌ها را در موقع لزوم می‌توان با محاسبه تابع بستار آیتم‌های هسته‌ای به‌دست آورد.

رسم دیاگرام و جداول تجزیه CLR و LALR

یک روش، برای تهیه دیاگرام LALR از طریق رسم دیاگرام CLR است. قبل توضیح این روش لازم است مفهوم آیتم‌های LR(1) تعریف گردد.

آیتم‌های LR(1): یک آیتم LR(1) یک زوج مرتب متشکل از یک آیتم LR(0) و یک مجموعه پایانه به نام مجموعه پیش‌بینی (Lookahead) است که معمولا به صورت $[A \rightarrow \alpha \bullet . LA]$ نمایش داده می‌شود. رابطه زیر در مورد مجموعه پیش‌بینی و مجموعه Follow غیرپایانه سمت چپ این آیتم‌ها برقرار است:

$$LA \subseteq Follow(A)$$

الگوریتم محاسبه تابع بستار که در مورد آیتم‌های LR(0) توضیح داده شده بسیار شبیه الگوریتم یافتن بستار آیتم‌های LR(1) است. با این تفاوت که در این‌جا باید به‌صورت زیر برای آیتم‌های غیرهسته‌ای جدیدی که به مجموعه بستار اضافه می‌شوند، مجموعه پیش‌بینی تعیین نمود:

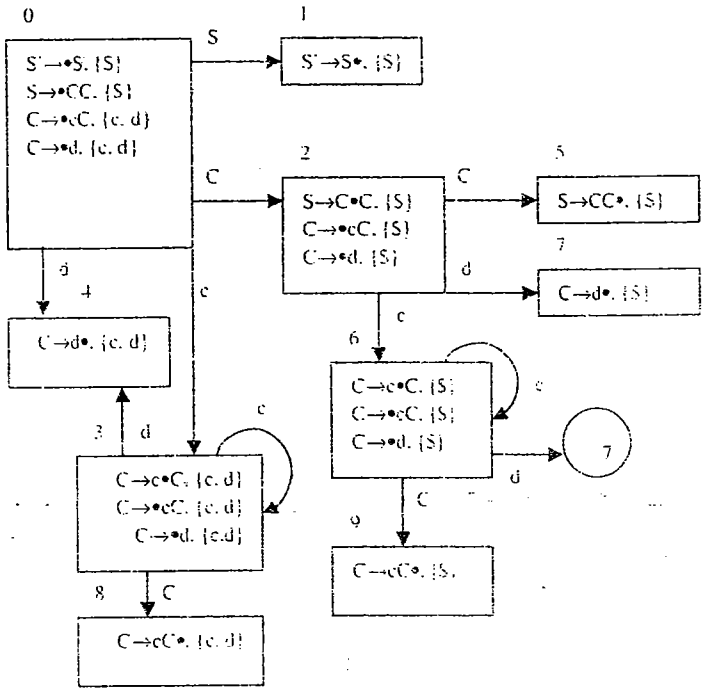
$$[A \rightarrow \alpha \bullet B\beta, \{a\}]$$
$$[B \rightarrow \bullet\delta, \{b\}] \mid b \in First(\beta a)$$

به عبارت ساده‌تر، مجموعه پیش‌بینی آیتم جدید $B \to \bullet\delta$ که به خاطر وجود آیتم $[A \to \alpha \bullet B\beta, \{a\}]$ به مجموعه بستار افزوده گردیده برابر است با مجموعه $First(\beta a)$.

طریقه رسم دیاگرام CLR مشابه روش رسم دیاگرام SLR است. با این تفاوت که در این‌جا در وضعیت‌های دیاگرام آیتم‌های LR(1) قرار داده می‌شود. به عنوان نمونه به گرامر زیر و دیاگرام CLR(1) آن که در ادامه آورده شده، توجه کنید.

1 $S \to CC$

2 – 3 $C \to cC \mid d$



نحوه تکمیل جدول تجزیه CLR(1)

روش تهیه جدول تجزیه CLR(1) بسیار شبیه روشی که در رابطه با SLR(1) بیان گردید با این تفاوت که اگر در این‌جا در وضعیت i آیتمی به فرم $[A \to \alpha\bullet, \{b\}]$ داشته باشیم، در خانه‌های action [i,b] دستور $A \to \alpha$ Reduce قرار می‌دهیم. به عبارت دیگر در این‌جا به‌جای استفاده از مجموعه‌های Follow از مجموعه‌های پیش‌بینی استفاده می‌گردد.

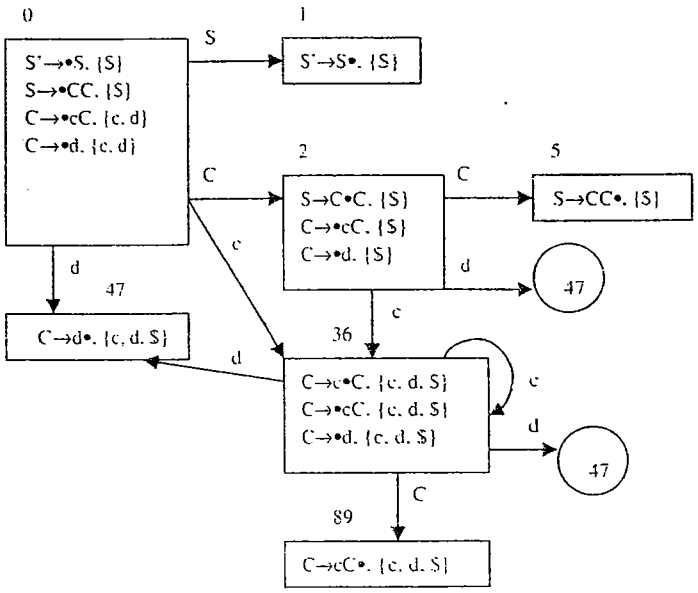
به عنوان یک نمونه جدول تجزیه CLR گرامر فوق به صورت زیر است:

	c	d	$	S	C
0	S_3	S_4		1	2
1			Acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

رسم دیاگرام و جدول تجزیه LALR

برای رسم دیاگرامر LALR یک گرامر باید ابتدا دیاگرام CLR گرامر را رسم کرد. سپس از روی دیاگرام CLR با انجام و عمل زیر دیاگرام LALR به‌دست می‌آید.

ابتدا در دیاگرام CLR به دنبال وضعیت‌هایی می‌گردیم که بخش آیتم‌های LR(0) آن‌ها (که با آن اصطلاحاً هسته یا core یک وضعیت می‌گویند) یکسان باشد. سپس در دیاگرام، وضعیت‌هایی را که هسته یکسانی دارند، ادغام می‌کنیم و مجموعه پیش‌بینی آیتم‌های این وضعیت‌ها را نظیر به نظیر با هم ادغام می‌کنیم. مثلاً دیاگرام فوق پس از ادغام وضعیت‌هایی که هسته‌های مشابهی دارند، به‌صورت زیر در می‌آید.



LR تداخل CLR دارد.

اگر گرامری CLR باشد و پس ادغام وضعیت‌های مزبور در جدول تجزیه LALR گرامر نیز تداخلی به‌وجود نیاید، می‌توان نتیجه گرفت که گرامر LALR است. حال اگر گرامری CLR بوده و LALR نباشد، پس از ادغام در جدول تجزیه LALR گرامر، تداخل نوع کاهش ـ کاهش بروز خواهد کرد. هیچ‌گاه در اثر این ادغام در جدول LALR تداخل نوع انتقال ـ کاهش به‌وجود نخواهد آمد، مگر آن‌که

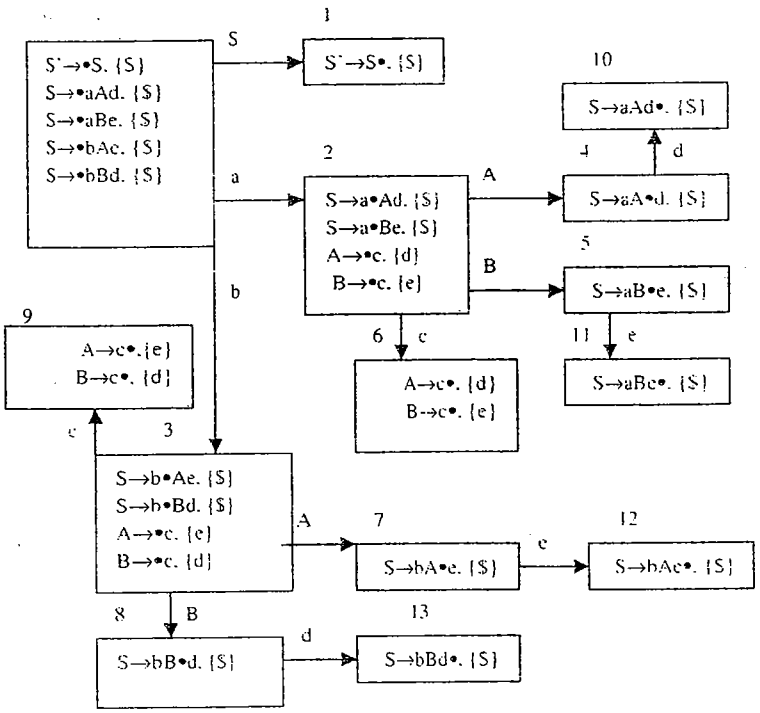
گرامر CLR هم نبوده باشد، که در این‌صورت قطعاً LALR هم نیست. به عبارت دیگر، در چنین حالتی، تداخل قطعاً قبل از ادغام نیز وجود داشته است.

طریقه تهیه جدول تجزیه LALR از روی دیاگرام ادغام شده، عیناً همانند روش تهیه جدول CLR است. به عنوان یک نمونه جدول تجزیه LALR مثال قبل به صورت زیر خواهد بود.

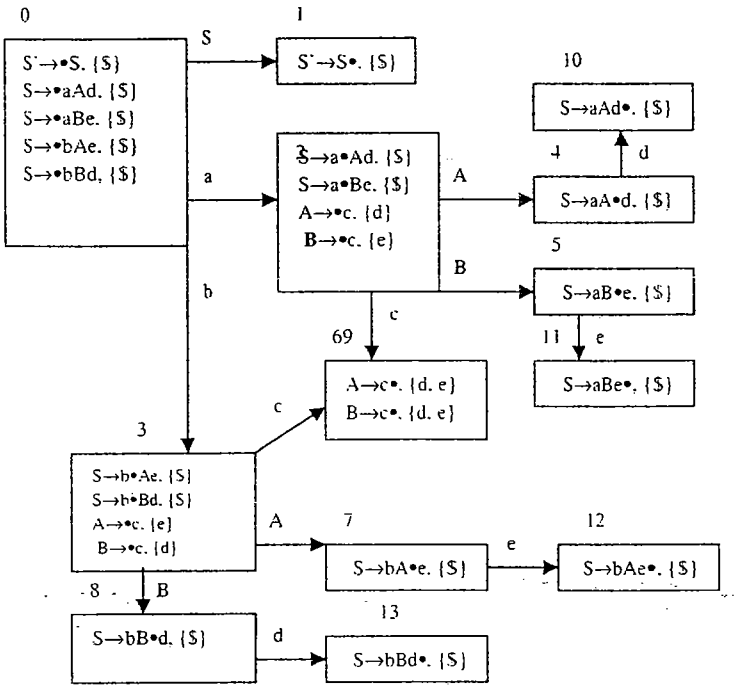
	C	d	$	S	C
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

حال به یک گرامر دیگر توجه کنید که LALR نیست. همان‌گونه که بیان شد ابتدا دیاگرام CLR گرامر را رسم کرده، از طریق آن دیاگرام و جدول تجزیه LALR گرامر را به‌دست می‌آوریم.

1 S → a A d
2 S → b B d
3 S → a B e
4 S → b A e
5 A → c
6 B → c



مشاهده می‌شود که در دیاگرام فوق وضعیت‌های 6 و 9 از نظر هسته یکسان‌اند. دیاگرام LALR گرامر کـه از ادغـام ایـن دو وضـعیت به‌دست می‌آید، به صورت زیر است:



هم‌چنین قسمتی از جدول تجزیه گرامر که از روی دیاگرام فوق به‌دست آمده به‌صورت زیر است:

	a	b	c	d	e	$	S	A	B
0	S_2	S_3					1		
1						acc			
2			S_{69}					4	5
3			S_{69}					8	7
4				S_{10}					
5					S_{11}				
69				r_5 / r_6	r_5 / r_6				

مشاهده می‌شود که: $action[69,e] = r_5 / r_6$ و $action[69,d] = r_5 / r_6$ است. بنابراین، گرامر موردنظر LALR نیست.

تعداد وضعیت‌های جداول تجزیه SLR و LALR هر گرامری دقیقاً یکسان است؛ لیکن تعداد وضعیت‌های جدول تجزیه CLR گرامرها معمولاً به مراتب بزرگتر از جداول LALR و SLR آن‌ها است. تنها عیب جزیی روش LALR نسبت به CLR این است که خطای‌های نحوی را ممکن است قدری دیرتر کشف نماید. البته، هیچ‌یک از این سه روش، پس از رسیدن به یک توکن غلط، آن‌را به داخل انباره انتقال نخواهد داد. در CLR پس از رسیدن به یک توکن خطا، دیگر حتی عمل کاهش نیز انجام نخواهد شد؛ لیکن در مورد SLR و LALR ممکن است خطا پس از اجرای چند عمل کاهش اضافی، کشف گردد.

به عنوان مثال، مجدداً جدول تجزیه گرامر ساده $C \rightarrow cC|d, S \rightarrow CC$ توجه کنید. فرم کلی رشته‌هایی که این امر توصیف می‌کند به‌صورت c*dc*d است. فرض کنید ورودی ما به‌صورت ccd باشد. عمل تجزیه این رشته را که دارای خطای نحوی است، به دو روش CLR و LALR مورد بررسی قرار می‌دهیم:

ابتدا در روش CLR قدم‌های طی شده تا مرحله کشف خطا به‌صورت زیر است:

انباره	ورودی	عمل انجام شده
0	ccd$	S3
0 c 3	cd$	S3
0 c 3 c 3	d$	S4
0 c 3 c 3 d 4	$	اعلام خطا

حال اگر با استفاده از جدول LALR گرامر عمل تجزیه انجام شود، این خطا پس از انجام سه قدم کاهش اضافی کشف می‌شود. این امر در شکل زیر نشان داده شده است.

انباره	ورودی	عمل انجام شده
0	c c d$	s 36
0 c 36	c d$	s 36
0 c 36c36	d$	s 47
0 c36 c d 47	$	r 3
0 c36 c36C 89	$	r 2
0 c36 C 89	$	r 2
0 C 2	$	اعلام خطا

مزیت استفاده از گرامرهای گنگ در روش LR

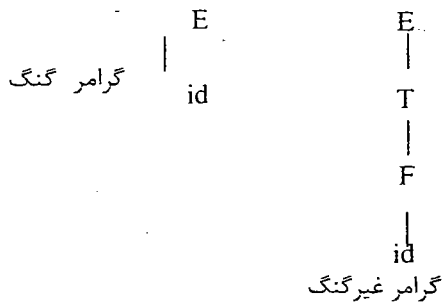
مجدداً گرامر گنگ زیر را که معرف عبارات جبری دارای عملگرهای + و * است، در نظر بگیرید:

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

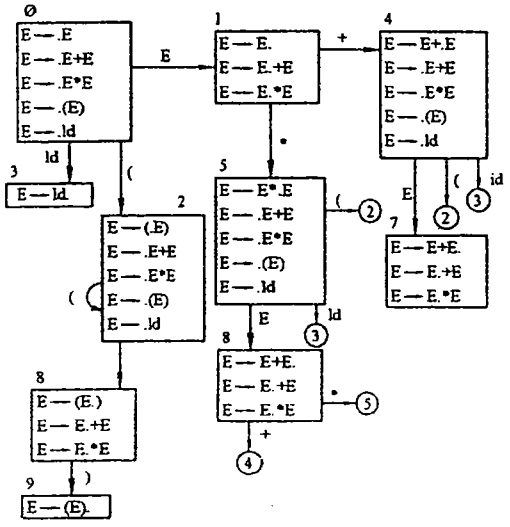
گرامر غیر گنگ معادل گرامر فوق به‌صورت زیر است:

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

در این گرامر، تقدم عملیاتی عملگرها نیز لحاظ شده است، در حالی‌که در گرامر قبلی همه عملگرها تقدم یکسانی دارند.
اشکال گرامرهای گنگ در رابطه با روش LR در این است که در جدول تجزیه، حتماً تداخل‌های انتقال / کاهش وجود خواهد داشت. حال، اگر بتوانیم به نوعی، این تداخل‌ها را رفع کنیم، استفاده از گرامر گنگ، ممکن است سبب افزایش کارآیی گردد. به عنوان نمونه، در صورت استفاده از گرامر گنگ فوق، هر توکن id تنها در یک مرحله به یک غیرپایانه E کاهش می‌یابد. این در حالی است که در گرامر غیر گنگ این عمل در سه قدم صورت می‌گیرد. این امر در شکل زیر نشان داده شده است.



خوشبختانه می‌توان با استفاده از اطلاعی که در مورد تقدم عملیات عملگرهای + و * داریم، این تداخل‌ها را برطرف کرد. به دیاگرام و جدول تجزیه SLR گرامر گنگ عبارات جبری که در ادامه آمده است، توجه کنید.



	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			acc	
2	S_3			S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		S_4, r_1	S_5, r_1		r_1	r_1	
8		S_4, r_2	S_5, r_2		r_2	r_2	
9		r_3	r_3		r_3	r_3	

برای رفع تداخل خانه action [7,+]، فرض کنید که رشته ورودی a+b+c باشد و در مرحله زیر از تجزیه باشیم.

انباره	ورودی	عملی که باید انجام شود
0	id + id + id$	s 3
0 id 3	+ id + id$	r 4
0 E1	+ id + id $	s 4
0 E 1 + 4	id + id $	s 3
0 E 1 + 4 id 3	+ id $	r 4
0 E 1 + 4 E 7	+ id $	بروز تداخل

در این لحظه، پارسر می‌تواند هم عمل انتقال و هم عمل کاهش از طریق قاعده شماره یک را انجام دهد. برای رفع این مشکل، چون تقدم عمل جمع، در یک عبارت از چپ به راست است، عمل کاهش انتخاب می‌شود. به این ترتیب، به عمل جمع سمت چپ تقدم بیش‌تری داده می‌شود. لازم به توضیح است که در روش‌های تجزیه پایین به بالا هر گاه که عمل کاهش از طریق یک قاعده صورت می‌گیرد، رویه تولید کد فراخوانی شده و کد مربوط به آن قاعده تولید می‌گردد. هم‌چنین در دستورات ترتیبی مثل عملیات جبری، کدی که زودتر تولید گردد، زودتر نیز اجرا خواهد شد. در مثال فوق، نیز به همین دلیل انتخاب عمل کاهش باعث می‌شود که کد عمل جمع سمت چپ زودتر تولید گردیده و در نتیجه تقدم بیش‌تری بیابد و زودتر از عمل جمع سمت راست اجرا گردد.

برای رفع تداخل خانه action [7,*] فرض کنید رشته ورودی a+b*c باشد. در این صورت پس از چند مرحله خواهیم داشت.

انباره	ورودی	عمل
0	id + id * id$	S_3
0 id 3	+ id * id$	r_4
0 E1	+ id * id $	S_4
0 E 1 + 4	id * id$	S_3
0 E 1 + 4 id3	* id $	r 4
0 E 1 + 4 E 7	* id $	تداخل روی می‌دهد

از آن‌جا که تقدم عملگر * بیش‌تر از + است و لذا ابتدا باید عمل ضرب انجام شود، عمل انتقال یعنی S5 انتخاب می‌گردد. به این ترتیب مطابق آن‌چه در بالا گفته شد، کد عمل * زودتر تولید و اجرا خواهد شد.
برای حذف تداخل action[4,+] جمله ورودی a*b+c فرض شده است.

انباره	ورودی	عمل
0	id * id + id$	S_3
0 id 3	*id + id$	r_4
0 E1	*id + id $	S_4
0 E 1*S	id+id$	S_3
0 E 1*S id3	+id $	r 4
0 E 1 *S E 8	+id $	تداخل روی می‌دهد

در این‌جا چون * نسبت به + مقدم است، عمل کاهش انتخاب می‌شود. به‌طور مشابه در رفع تداخل خانه action [8,*] نیز باید عمل کاهش انتخاب گردد.

اصلاح خطا در روش LR

یک پارسر LR زمانی متوجه یک خطا می‌شود که در هنگام مراجعه به بخش action جدول تجزیه به یک خانه خالی رجوع کند. دو روش اصلاح خطایی که در مورد روش تجزیه LL(1) معرفی گردید، در روش LR(1) نیز قابل به‌کارگیری هستند. این روش‌ها، عبارت بودند از روش Panic Mode و روش Phrase Level. روش اول به‌صورت زیر پیاده‌سازی می‌گردد:
در موقع برخورد با یک خطای نحوی، پارسر داخل انباره آن‌قدر پایین می‌رود تا به حالتی مثل S برخورد کند به‌طوری‌که برای غیر پایانه‌ای مثل A در خانه goto[S,A] مقداری مثل n وجود داشته باشد. سپس، پارسر عناصر بالای انباره تا S را حذف نموده، n و A را به ترتیب وارد انباره می‌کند. ضمناً از کارکترهای ورودی نیز آن‌قدر حذف می‌کند تا به یکی از عناصر مجموعه Follow (A) برسد. این روش پیاده‌سازی آسانی دارد، لیکن ممکن است بخش زیادی از ورودی، تا رسیدن به عنصر مزبور بدون بررسی نحوی حذف گردد.
در روش دیگر، یعنی Phrase Level، هیچ بخشی از ورودی بدون بررسی حذف نمی‌شود. این روش، خطا را در همان محل که اتفاق افتاده اصلاح می‌کند. برای این‌کار، بایستی در خانه‌های خالی جدول action نشانه‌روهایی به رویه‌های مناسب اصلاح خطا قرار داد تا در صورت مراجعه به این خانه‌ها از طریق نشانه‌روها رویه مربوطه فعال گردد:
مثلاً، جدول تجزیه زیر که همان جدول تجزیه گرامر مثال قبل است، بسته به نوع خطا در خانه‌های خالی نشانه‌روهایی به‌صورت ei به رویه مناسب اصلاح خطا قرار داده شده است. یک نکته مهم در مورد هر دو این روش‌های اصلاح خطا (به‌ویژه روش دوم) این است که امکان دارد در اثر اصلاح مکرر خطا، برنامه در یک حلقه بی‌انتها گرفتار شود، لذا لازم است که طراح کامپایلر تدابیری جهت پیش‌گیری از این امر بیاندیشد.

	id	+	*	(	)	$	E
0	S_3	e_1	e_1	S_2	e_2	e_1	1
1	e_3	S_4	S_5	e_3	e_2	acc	
2	S_3	e_1	e_1	S_2	e_2	e_1	6
3	r_4	r_4	r_4	r_4	r_4	r_4	
4	S_3	e_1	e_1	S_2	e_2	e_1	7
5	S_3	e_1	e_1	S_2	e_2	e_1	8
6	e_3	S_4	S_5	e_3	S_9	e_4	
7	r_1	r_1	S_5	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	r_2	r_2	
9	r_3	r_3	r_3	r_3	r_3	r_3	

e1: یک id و سپس وضعیت ۳ را بر روی انباره قرار بده و پیغام "عملوند گمشده" را چاپ کن.

e2: پرانتز را از ورودی حذف کن و پیغام "پرانتز بسته زیادی" را چاپ کن.

e3: یک عملگر + و سپس وضعیت ۴ را به روی انباره قرار بده و پیغام "پرانتز بسته گمشده" را چاپ کن.

به عنوان مثال، به تجزیه عبارت (id+ با استفاده از جدول فوق توجه کنید:

انباره	ورودی	عمل
0	id+)$	S_3
0 id 3	+)$	r_4
0 E1	+)$	S_4
0 E 1+4	)$	e_2: پرانتز بستهٔ زیادی
0 E 1+4	$	e_1: عملوند گمشده
0 E 1+4 id3	$	r_4
0 E 1+4 E7	$	r_1
0 E 1	$	

استراتژی‌های تخصیص حافظه

یکی از اعمالی که کامپایلر انجام می‌دهد، تخصیص حافظه مورد نیاز برنامه در زمان اجرا است. البته تخصیص واقعی حافظه زمان اجرا توسط سیستم عامل و در همان زمان اجرا صورت خواهد گرفت. لیکن، این کامپایلر است که در زمان کامپایل مشخص می‌کند که کد تولید شده به چه مقدار حافظه برای ا جرا نیاز دارد. آدرس‌هایی که کامپایلر به متغیرها اختصاص می‌دهد، معمولاً آدرس‌های نسبی است که نسبت به یک آدرس صفرمجازی در نظر گرفته می‌شود. کامپایلر فرض می‌کند که حافظه کامپیوتر مقصد، نامحدود است و آدرس شروع آن مثلاً، عدد صفر است. سپس در زمان اجرا، این سیستم عامل است که قبل از اجرای یک برنامه، آدرس‌هایی را که کامپایلر به برنامه اختصاص داده را اصلاح می‌کند (مثلاً بااضافه کردن مقدار ثابتی به همه آدرس‌ها). به این نوع کد، کد جابه‌جاپذیر

(Relocateable Code) گویند. کامپایلرهای مختلف به روش‌های مختلفی، حافظه زمان اجرا را تخصیص می‌دهند که مهم‌ترین آن‌ها عبارتند از:

۱ـ تخصیص ایستا (Static Allocation)

۲ـ تخصیص انباره‌ای (Stack Allocation)

۳ـ تخصیص توده (Heap Allocation)

تعاریف اولیه:

فعالیت (Activation) : به هر مرتبه اجرای یک رویه یک فعالیت آن رویه گویند.

طول عمر (Life time): هر رویه‌ای دارای یک طول عمر است که عبارت است از مجموعه قدم‌هایی که در موقع اجرای آن رویه انجام می‌شود. به عبارتی دیگر، سلسله قدم‌هایی را که بین آغاز و انتهای اجرای یک رویه مثل P انجام می‌شود، طول عمر آن رویه گویند.

درخت فعالیت (Activeation Tree): درخت فعالیت یک برنامه نشان می‌دهد که چگونه یا چه موقع، کنترل از یک رویه به یک رویه دیگر انتقال می‌یابد. به عبارت دیگر، مسیر حرکت کنترل از یک رویه به رویه دیگر را به‌وسیله ساختاری به نام درخت فعالیت، نشان می‌دهیم.

یک درخت فعالیت دارای مشخصات زیر است:

۱ـ هر گره این درخت اجرای یک رویه را نشان می‌دهد.

۲ـ گره a، پدر گره b محسوب می‌شود، اگر و فقط اگر جریان کنترل از اجرای a به اجرای b منتقل شود.

۳ـ ریشه درخت اجرای برنامه اصلی را نشان می‌دهد.

۴ـ گره a سمت چپ گره b قرار می‌گیرد، اگر وفقط اگر عمر a قبل از شروع عمر b ظاهر شود.

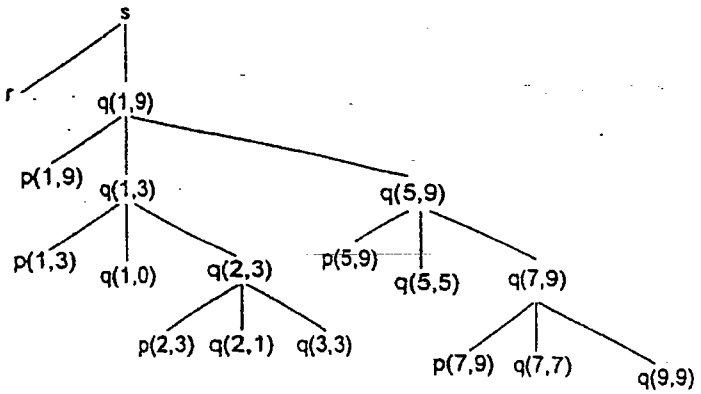
به عنوان مثال ، برنامه sort زیر را در نظر بگیرید:

```
Program sort (input, output);
    var a :array [0..10] of integer
     procedure readarry;
          var i:integer;
           begin
                 for i:=1  to  10  do read (a[i]);
           end;
function partition (y, z:integer):integer;
           var i,j,x,v:integer;
            begin..
            end;
procedure quicksort (m,n:integer);
           var i: integer;
            begin
              if  (n>m)then begin
              i:=partition (m,n);
              quicksort(m,i-1);
               quicksort (i+1,n);
             end
      end;
begin
       a[0]:=-9999;a[10]:=9999;
        readarray;
        quicksort(1,9);
end.
```

برای رسم درخت فعالیت، یک دستور Print درا بتدا و انتهای رویه‌ها قرار می‌دهیم تا پیام‌های مناسبی چاپ کند. به این ترتیب خروجی زیر فعالیت‌های رویه‌ها را نشان می‌دهد.

```
Execution Begins....
enter readarray
leave readarry
enter quicksort (1,9)
enter partition (1,9)
leave partition (1.9)
enter quicksort (1,3)
    ...
leave quicksort (1,3)
enter quicksort(5,9)
    ...
leave quicksort (5,9)
leave quicksort (1.9)
execution terminated
```

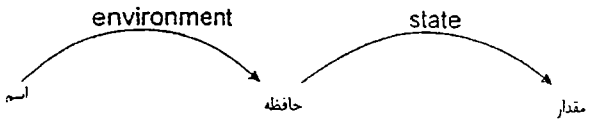
با توجه به خروجی فوق درخت فعالیت (Activation Tree) به‌صورت زیر رسم می‌شود.



انباره کنترل (Control-Satck): انباره‌ای است که از آن برای نگهداری ردپای مسیر جریان کنترل از یک رویه به رویه دیگر استفاده می‌شود. به این ترتیب که با فعال شدن یک رویه، اسم آن رویه به بالای انباره وارد شده و هر گاه اجرای رویه به پایان رسید، اسم آن نیز از بالای انباره حذف می‌شود.

محیط (Environment) : تابعی است که یک خانه از حافظه را به یک اسم مرتبط می‌کند.

حالت (State): تابعی است که یک خانه از حافظه را به یک مقدار متصل می‌کند. رفتار توابع فوق را می‌توان با شکل زیر نشان داد.



این دو تابع با یکدیگر متفاوتند. به عنوان مثال دستور انتساب (assignment) تنها تابع حالت را تغییر می‌دهد و بر روی تابع محیط اثری ندارد.

روشی که توسط آن کمپایلر یک زبان باید حافظه زمان اجرا را سازماندهی کند و اسامی را به حافظه‌ها مربوط کند، تا حد بسیار زیادی تحت تأثیر جواب سؤالات زیر واقع می‌شود:

۱ـ آیا رویه‌ها می‌توانند بازگشتی باشند؟

۲ـ مقادیر متغیرهای محلی پس از بازگشت کنترل از یک رویه چه می‌شوند؟

۳ـ آیا رویه‌ها می‌توانند به متغیرهای سراسری رجوع کنند؟

۴ـ نحوه انتقال پارامترها در هنگام فراخواندن یک رویه به چه شکل است؟

۵ـ آیا خود رویه‌ها هم می‌توانند به صورت پارامتر انتقال یابند؟

۶ـ آیا خود رویه‌ها می‌توانند به صورت نتیجه برگردانده شوند؟

۷ـ آیا می‌توان حافظه را در زمان اجرا به صورت پویا اختصاص داد؟

۸ـ آیا حافظه‌های اختصاص داده شده را بایستی به صورت صریح آزاد نمود؟

حافظه‌ای که کمپایلر برای اجرای یک برنامه اختصاص می‌دهد شامل موارد زیر است:

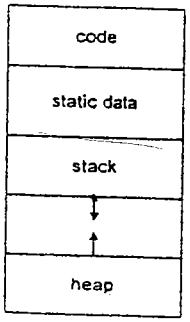
۱ـ کد تولید شده مقصد

۲ـ داده‌های برنامه

۳ـ انباره‌ای برای کنترل اجرای رویه‌ها

۴ـ قسمتی به صورت توده heap برای حافظه‌هایی که در زمان اجرای برنامه باید تخصیص داده شوند.

شکل زیر یک سازماندهی نوعی از حافظه زمان اجرا را نشان می‌دهد که در آن اندازه بخش‌های heap و stack می‌تواند در زمان اجرای تغییر کند.



رکود فعالیت (Activation Record) :برای نگهداری اطلاعاتی که جهت کنترل اجرای رویه‌ها لازم‌اند، کمپایلر از رکوردی به نام رکورد فعالیت استفاده می‌کند. رکورد فعالیت را قاب (Frame) نیز می‌نامند. این رکورد نوعاً دارای فیلدهایی زیر است:

return value
actual parameters
optional control link
optional access link
saved machine status
local data
temporaries

رکوردهای فعالیت در همه کمپایلرها دارای فرمت فوق نیستند. اندازه هر یک از فیلدهای فوق در زمان کامپایل دستور فراخوانی رویه‌ها تعیین می‌گردد. به عبارت دیگر، اندازه تقریبا تمام این فیلدها در زمان کمپایل مشخص می‌شود؛ به‌جز مواردی از قبیل آن‌که رویه‌ای دارای یک آرایه‌ای محلی است که اندازه آن آرایه از طریق انتقال یک آرگومان به آن رویه تعیین می‌گردد. در این مورد اندازه حافظه موردنیاز زمانی که رویه در زمان اجرا فراخوانده می‌شود، تعیین می‌گردد.

در روش تخصیص حافظه به صورت ایستا، رکوردهای فعالیت در بخش static data ذخیره می‌شوند. توجه کنید که هر رویه، یک رکورد فعالیت خاص خود دارد. مورد استفاده این فیلدها به قرار زیر است:

۱ـ Temporaries: این فیلد، شامل حافظه‌های موقتی است که جهت نگهداری نتایج بینابینی محاسبات موردنیاز است. به عبارتی هر رویه‌ای برای اجرا نیاز به یک سری حافظه موقتی دارد که این حافظه از طریق این فیلد تأمین می‌گردد.

۲ـ Local Data: هر رویه‌ای دارای یک سری متغیرهای محلی است. برای این متغیرها حافظه‌ای در این فیلد در نظر گرفته می‌شود.

۳ـ Saved Machine Status: در این فیلد، اطلاعاتی درباره وضعیت ماشین، قبل از این که رویه فراخوانی شود، نگهداری می‌گردد. این اطلاعات، شامل مقادیری نظیر شمارنده برنامه (Program Counter) و ثبات‌های مهم ماشین می‌شود. از این مقادیر، جهت اعاده وضعیت ماشین و بازگرداندن کنترل به محل مناسب، بعد از اتمام کار رویه، استفاده می‌گردد.

۴ـ Optional Access Link: این فیلد، یک نشانه‌رو است که از آن برای دست‌یابی به متغیرهای سراسری استفاده می‌شود. این نشانه‌رو، به رکورد فعالیت نزدیک‌ترین رویه‌ای اشاره می‌کند که متغیرهای آن برای رویه فعلی، سراسری محسوب می‌شوند.

۵ـ Optional Control Link: این نشانه‌رویی است که به رکورد فعالیت رویه فراخواننده رویه جاری اشاره می‌کند.

۶ـ Actual Parameter: پارامترهای واقعی در این بخش قرار می‌گیرند. به عبارت دیگر، رویه فراخواننده، پارامترهای واقعی را از طریق این محل در اختیار رویه فراخوانده شده قرار می‌دهد.

۷ـ Return Value: بعد از پایان یافتن رویه فعلی، مقادیری که باید به رویه فراخواننده بازگردانده شوند، در این محل قرار می‌گیرند.

مقدار حافظه موردنیاز برای یک متغیر از روی نوع آن (Type) مشخص می‌شود. در برخی موارد، ممکن است نیاز به در نظر گرفتن مسأله‌ای به نام به‌خط شدن (Alignment) هم پیش بیاید. یعنی اگر محدودیتی در نحوه حافظه‌دهی به متغیرها وجود داشته باشد، این محدودیت‌ها هم در میزان حافظه تخصیص داده شده مؤثر است. مثلاً اگر در ماشینی آدرس عملوندها حتماً باید مضربی از ۴ باشد، ممکن است مقداری از حافظه به این ترتیب هدر رود. به عنوان مثال، یک آرایه ۱۰ کارکتری تنها به ۱۰ بایت حافظه احتیاج دارد ولی به خاطر محدودیت قید شده کامپایلر مجبور است ۱۲ بایت به این آرایه اختصاص دهد، که به این ترتیب ۲ بایت اضافی بی‌استفاده خواهد بود. در این چنین حالات، به مقدار حافظه اضافی که تنها به‌خاطر مسأله به خط شدن به متغیری اختصاص داده شده است، Padding گویند. برای رفع نیاز به Padding، کامپایلر می‌تواند اطلاعات را فشرده (Packed) کند که در این صورت، موقع استفاده ناچار است، دوباره با برگرداندن متغیر به فرم عادی (Unpacking)، مقدار اصلی متغیر را به‌دست آورد.

استراتژی‌های تخصیص حافظه

همان‌گونه که در بالا اشاره گردید، سه روش متداول برای تخصیص حافظه زمان اجرا وجود دارد، که در ادامه به شرح هر یک می‌پردازیم. این روش‌ها عبارتند از تخصیص ایستا، انباره‌ای و توده. ابتدا روش تخصیص حافظه به صورت ایستا که ساده‌ترین روش تخصیص حافظه زمان اجرا است، معرفی می‌گردد.

تخصیص حافظه به روش ایستا (Static Allocation)

در این روش، به هر متغیر یک خانه از حافظه اختصاص داده می‌شود و از اول تا پایان اجرای برنامه اصلی، این خانه از حافظه در اختیار آن متغیر باقی می‌ماند. به عبارتی دیگر، در تمام طول اجرای برنامه این حافظه به آن متغیر تعلق خواهد داشت. در صورتی که لازم باشد، حتی بعد از پایان یافتن رویه‌ها نیز مقدار متغیرهای محلی‌شان حفظ شود، باید از تخصیص حافظه به روش ایستا استفاده کرد. به عنوان نمونه، کامپایلر زبان فرترن به روش ایستا حافظه تخصیص می‌دهد. کامپایلری که فقط از این روش تخصیص حافظه استفاده کند، دارای محدودیت‌های زیر است:

۱ـ اندازه داده‌ها و محدودیت‌هایی که روی نحوه آدرس‌دهی وجود دارد، بایستی کاملاً در زمان کمپایل مشخص باشد.

۲ـ نمی‌توان برنامه‌های بازگشتی داشت، زیرا همه اجراهای یک رویه از محیط یکسانی استفاده می‌کنند. به عبارتی دیگر، برای تخصیص حافظه به متغیرهای محلی از یک خانه حافظه استفاده می‌کنند.

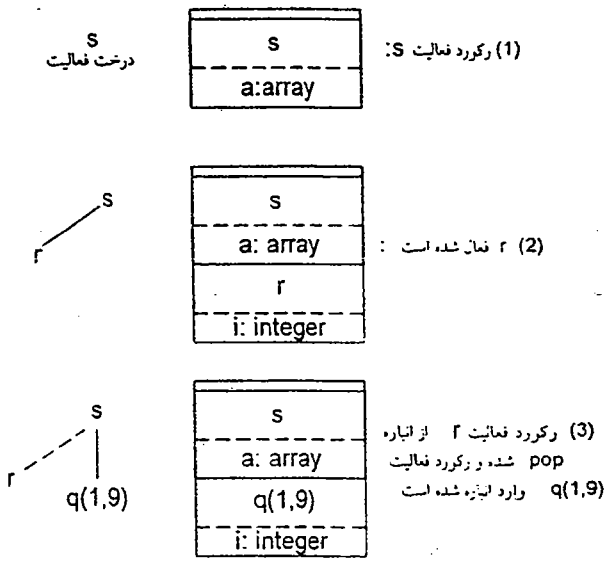
۳ـ ساختارهای داده‌ای را نمی‌توان به صورت پویا ایجاد کرد؛ یعنی هیچ مکانیزمی جهت تخصیص حافظه در زمان اجرا وجود ندارد.

تخصیص حافظه به روش انباره‌ای (Stack Allocation)

در این نوع تخصیص حافظه، از روش کنترل انباره استفاده می‌شود. به این ترتیب که حافظه به صورت یک انباره سازماندهی می‌شود. هر بار که رویه‌ای فراخوانی می‌شود، رکورد فعالیت آن وارد انباره می‌شود و در پایان فعالیت آن رویه، رکورد فعالیتش از بالای انباره حذف می‌گردد. به این ترتیب حافظه‌های لازم برای متغیرهای محلی در هر بار فراخوانی یک رویه از طریق رکورد فعالیت آن رویه تخصیص داده می‌شوند. بنابراین، هر بار فراخوانی یک رویه، یک سری خانه جدید برای متغیرهای محلی آن رویه در نظر گرفته می‌شود و با پایان کار آن رویه، مقادیر متغیرهای محلی آن رویه نیز از بین می‌رود.

اگر از نشانه‌رویی به نام Top برای مشخص کردن بالای انباره استفاده کنیم، در زمان اجرا به وسیله کم و زیاد کردن مقدار top به اندازه طول رکورد فعالیت، می‌توان یک رکورد فعالیت را وارد انباره و یا از آن خارج کرد. به این ترتیب که اگر q رویه‌ای با رکورد فعالیتی به اندازه a باشد، قبل از آغاز اجرای q مقدار top به top+a تغییر پیدا می‌کند و زمانی که کنترل از q باز می‌گردد، top به top-a تغییر داده خواهد شد.

به عنوان مثال، درخت فعالیت مربوط به رویه‌های برنامه quicksort را در نظر بگیرید. برای این مثال وضعیت stack مطابق شکل زیر خواهد بود:

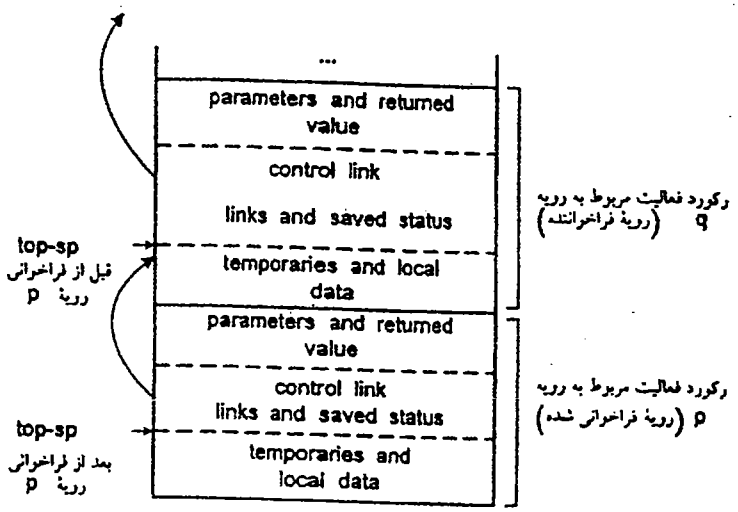


دنباله فراخوانی (Calling Sequence): عبارت است از یک سری دستورالعمل که در هنگام فراخوانی یک رویه انجام گرفته و باعث ایجاد رکورد فعالیت آن رویه و قرار گرفتن اطلاعات لازم در فیلدهای مربوط این رکورد می‌شوند.

دنباله بازگشت (Return Sequence): عبارت است از دستورالعمل‌هایی که در هنگام بازگشت از یک رویه باید انجام گیرند. این دنباله و دنباله فراخوانی توأماً وظیفه کنترل فراخوانی‌ها را بر عهده دارند.

وظایفی که دنباله‌های فراخوانی و بازگشت باید انجام دهند، بین رویه فراخواننده و رویه فراخوانی شده تقسیم می‌شود. به این ترتیب که بخشی از وظایف را رویه فراخواننده و مابقی را رویه فراخوانی شده انجام می‌دهند. ممکن است این سؤال پیش بیاید که بر اساس چه الگویی، این وظایف بین دو رویه مزبور تقسیم می‌گردد. با توجه به این که اگر رویه P به تعداد n مرتبه فراخوانی شود، بخشی از وظایف که به فراخواننده محول می‌گردد باید n مرتبه تولید گردد، اما آن بخش از دنباله فراخوانی که خود p باید انجام دهد فقط یک بار ایجاد می‌گردد، بهتر است که بیش‌تر وظایفی را که دنباله فراخوانی باید انجام دهد، در صورت امکان بر عهده برنامه فراخوانی شده محول نمود.

به عنوان نمونه‌ای از تقسیم‌بندی وظایف دنباله فراخوانی از بین رویه فراخوانی شده و رویه فراخواننده، فرض کنید رویه‌ای بنام q، رویه‌ای به‌نام p را فراخوانی کرده است. هم‌چنین فرض کنید یک نشانه‌رو بنام top-sp به انتهای فیلد وضعیت ماشین در رکورد فعالیت رویه q اشاره می‌کند. این مکان برای q مشخص است و می‌تواند آن‌را قبل از این که کنترل به رویه p (رویه فراخوانی شده) منتقل گردد، به نحوی ذخیره کند.



نمونه‌ای از تقسیم وظایف دنباله فراخوانی:

۱ـ فراخواننده مقدار پارامترهای واقعی را محاسبه کرده ودر فیلد مربوط به رویه فراخوانی شده قرار می‌دهد.

۲ـ فراخواننده یک آدرس بازگشت ومقدار قبلی Top-sp را در رکورد فعالیت فراخوانده شده ذخیره می‌کند. سپس Top-sp را به مکان جدیدش که در شکل نشان داده شده مقداردهی می‌کند.

۳ـ فراخوانده شده، مقادیر ثبات‌ها و اطلاعات لازم در مورد وضعیت ماشین را ذخیره می‌کند.

۴ـ فراخوانده شده به داده‌های محلی اش مقدار اولیه داده و اجرا را شروع می‌کند.

نمونه‌ای از تقسیم وظایف در دنباله بازگشت:

۱ـ فراخوانده شده نتیجه را نزدیک رکورد فعالیت فراخواننده قرار می‌دهد.

۲ـ فراخوانده شده با استفاده از اطلاعات فیلد وضعیت ماشین خود، مقدار قبلی top-sp و ثبات‌ها را اعاده می‌کند وکنترل رابه آدرس بازگشت در فراخواننده باز می‌گرداند.

۳ـ فراخواننده، مقدار بازگشتی را در رکورد خود کپی می‌کند. توجه داشته باشید با این‌که مقدار top-sp کاهش داده شده، فراخوانده به مقدار نتیجه بازگشتی دسترسی دارد و می‌تواند آن را در رکورد فعالیت خود کپی کند.

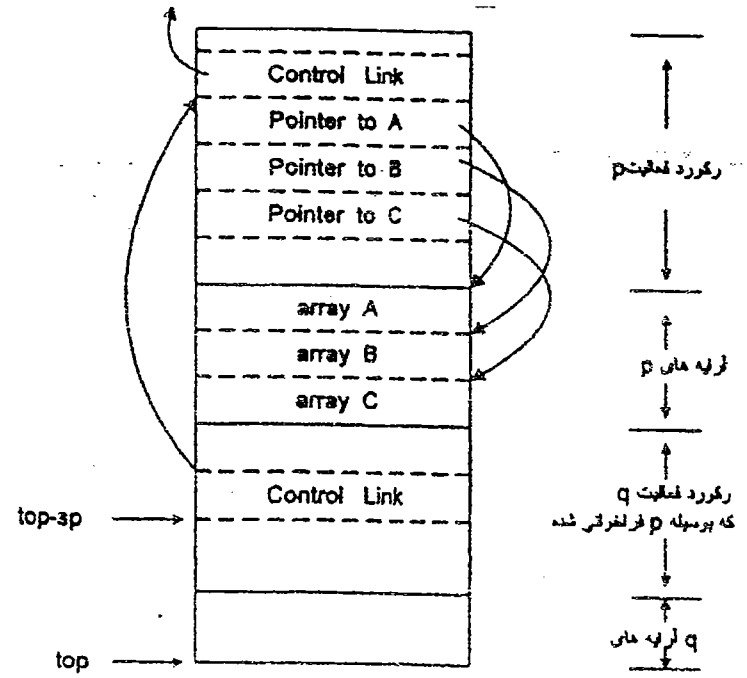
آدرس سرگردان (Dangling Reference) : زمانی که یک خانه از حافظه، امکان آزاد شدن داشته باشد، مسأله آدرس‌های سرگردان پیش می‌آید. اگر در یک برنامه به خانه‌ای از حافظه که قبلاً آزاد شده است، رجوع شود، به آدرس حافظه آزاد شده، آدرس سرگردان می‌گویند. این در واقع یک خطای منطقی است که اصولاً توسط کمپایلر تشخیص داده نمی‌شود و در زمان اجرا مشخص می‌گردد. به عنوان نمونه برنامه صفحه بعد را در نظر بگیرید:

```
main( )
{
    int *p;
    p = dangle ( )
}
int*dangle ( )
{
    int i=23;
    return &i;
}
```

با بازگشت از رویه ( ) dangle خانه i آزاد می‌شود. بنابراین p اشاره به خانه‌ای دارد که اصلاً وجود ندارد. لذا، در این‌جا مسأله آدرس سرگردان پیش آمده و در صورت استفاده از p نتیجه اشتباه به‌دست خواهد آمد.

داده‌های با طول متغیر (Variable Length Data): در زمان کمپایل، طول اکثر فیلدهای رکورد فعالیت رویه مشخص است. بنابراین، طول خود رکورد فعالیت نیز مشخص است. در صورتی که در رویه‌ای آرایه‌ای با طول نامشخص وجود داشته باشد، طول رکورد فعالیت، کاملاً مشخص نخواهد بود. در این حالت، کامپایلر برای نشان دادن آدرس شروع این آرایه‌ها از نشانه روهایی استفاده می‌کند تا در زمان اجرا اندازه این نوع آرایه‌ها مشخص شود.

به عنوان مثال، در شکل زیر رویه p دارای سه آرایه محلی است. برخلاف سایر متغیرهای محلی، حافظه اختصاص یافته به این سه آرایه درون رکورد فعالیت p نخواهد بود. بلکه، تنها یک نشانه رو، به آغاز هر آرایه در رکورد فعالیت p در نظر گرفته می‌شود. در زمان اجرا با استفاده از این نشانه‌روها می‌توان به عناصر آرایه‌ها دسترسی داشت. در این مثال، رویه q به‌وسیله رویه p فراخوانی شده است. در این صورت رکورد فعالیت q بعد از آرایه‌های p قرار می‌گیرند و به همین ترتیب، آرایه‌های با طول متغیر رویه q به دنبال رکورد فعالیت q واقع می‌شوند.



جهت دست‌یابی به محتویات انباره، از دو نشانه رو top و top-sp استفاده می‌شود. نشانه روی اول، همواره به بالای انباره اشاره می‌کند؛ در واقع به محلی اشاره می‌کند که رکورد فعالیت بعدی باید از آن‌جا آغاز شود. دومی، که جهت یافتن داده‌های محلی به کار می‌رود وبه انتهای فیلد وضعیت ماشین اشاره می‌کند. (در شکل فوق ، top-sp به این فیلد از رکورد در فعالیت q اشاره می‌کند.) درون این فیلد یک پیوند کنترل (Control Link) به مقدار قبلی top-sp (مربوط به زمانی که کنترل در رویه p بود) قرار دارد. در زمان بازگشت از q، مقدار جدید top به‌صورت زیر محاسبه می‌گردد:

top=top-sp- (مجموع اندازه فیلد پارامترها و وضعیت ماشین رکورد فعالیت q)

پس از محاسبه مقدار top ، مقدار جدید top-sp را نیز می‌توان از روی فیلد پیوند کنترل رویه q کپی می‌شود.

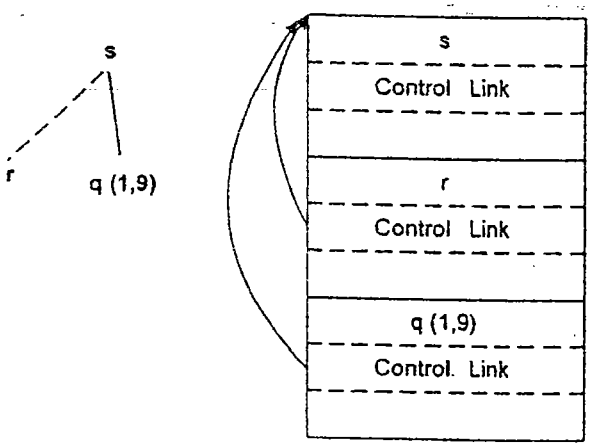
تخصیص حافظه به روش توده (Heap)

در صورتی که یکی از شرایط زیر موجود باشد، نمی‌توان از روش انباره برای تخصیص حافظه استفاده کرد:

۱ـ در صورتی که لازم باشد که متغیرهای محلی رویه‌ها مقادیر خود را پس از خاتمه اجرای آن رویه‌ها نیز حفظ کنند. در این حالت، می‌توان از روش‌های توده و یا ایستا استفاده کرد.

۲ـ این امر، مجاز باشد که اجرای رویه فراخوانی شده، دیرتر از رویه فراخواننده خاتمه یابد. این حالت در زبان‌هایی که در آن‌ها درخت فعالیت، منعکس کننده مسیر جریان کنترل بین رویه‌ها است، رخ نمی‌دهد.

در روش توده، مقدار حافظه موردنیاز به صورت حافظه‌های متوالی و بلوک شده در اختیار برنامه‌ها قرار می‌گیرد. تفاوت بین دو روش تخصیص توده‌ای و انباره‌ای در مدیریت رکوردهای فعالیت با استفاده از شکل زیر مشخص می‌شود. فرض کنید لازم باشد رکورد فعالیت r بعد از اتمام کار این رویه هم‌چنان باقی بماند. در این صورت رکورد فعالیت رویه q نمی‌تواند بلافاصله پس از رکورد فعالیت s قرار بگیرد. حال، اگر پس از ایجاد رکورد فعالیت q ، بخواهیم رکورد فعالیت r را حذف کنیم، یک فضای خالی بین رکورد فعالیت s و رکورد فعالیت q به‌وجود می‌آید و این وظیفه برنامه مدیریت توده است که به نحوی از این فضا استفاده کند. مدیریت توده، برنامه‌ای است که حافظه توده را کنترل کند؛ به این ترتیب که تعیین می‌کند چه حافظه‌ای به چه متغیری اختصاص یابد، به‌طوری‌که حافظه اضافی مصرف نگردد.



روشی برای بهبود کارآیی تخصیص حافظه به‌صورت توده

جهت بهبود کارایی روش تخصیص توده‌ای حافظه، می‌توان به صورت زیر عمل نمود:

۱ـ برای هر اندازه مورد نیاز یک لیست پیوندی از آدرس بلوک‌های حافظه به اندازه موردنظر، نگهداری می‌کنیم.

۲ـ در صورت امکان، یک تقاضا برای اندازه S را با یک بلوک به اندازه S' که در آن S' کوچکترین اندازه بزرگتر یا مساوی با S است، تامین می‌کنیم. هنگامی که یک بلوک آزاد می‌شود، آدرس شروع آن را به لیست پیوندی مربوطه‌اش اضافه می‌کنیم.

۳ـ برای بلوک‌های وسیعتر که عمل فوق امکانپذیر نیست، برنامه مدیریت توده را فراخوانی می‌کنیم که حافظه مورد درخواست را اختصاص دهد.

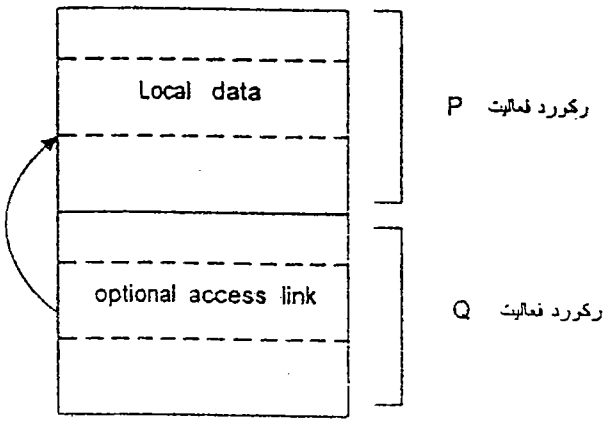
نحوه دست‌یابی به متغیرهای سراسری

برای دست‌یابی به متغیرهای سراسری دو روش وجود دارد:

۱ـ استفاده از فیلد پیوند دسترسی (Access Link) در رکورد فعالیت رویه‌ها

۲ـ استفاده از حافظه موسوم به Display

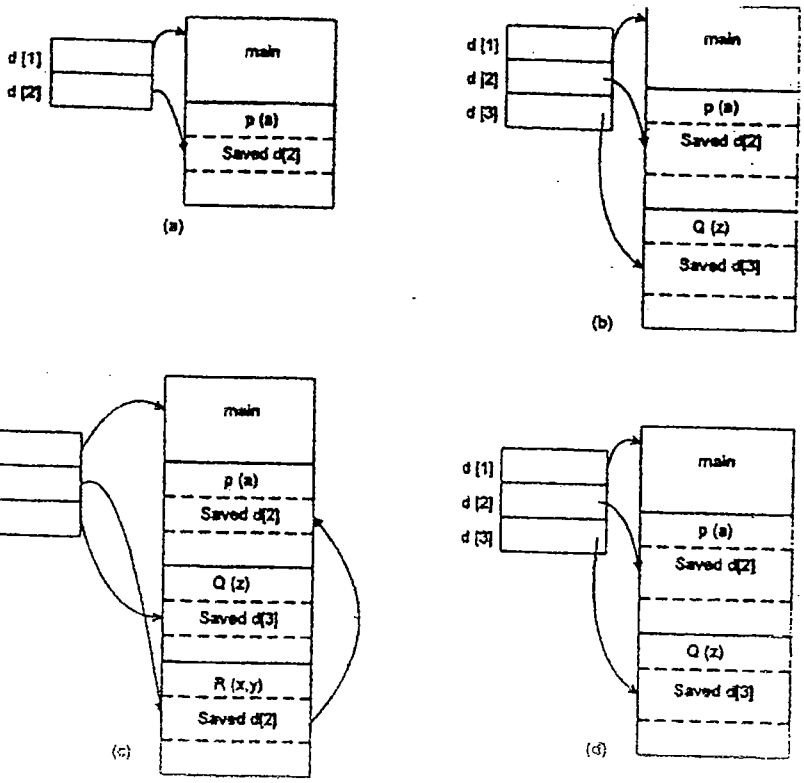
در روش اول، اگر رویه p، متغیرهایش برای رویه q سراسری محسوب شوند، از فیلد پیوند دسترسی رکورد q نشانه‌رویی، مطابق شکل زیر، به فیلد داده‌های محلی (Local Data) رکورد فعالیت رویه p اشاره می‌کند. در این روش، ممکن است برای دسترسی به برخی از متغیرهای سراسری، لازم باشد که چندین پیوند دسترسی پیمایش شوند. فرضاً اگر متغیرهای رویه r برای رویه p و متغیرهای رویه p برای رویه q سراسری محسوب شود، و اگر بخواهیم از درون رویه q به متغیری که در رویه r تعریف شده، مراجعه نماییم، بایستی دو پیوند دسترسی پیمایش گردد.



در روش دوم، به منظور دست‌یابی به متغیرهای سراسری، از آرایه‌ای از نشانه‌روها استفاده می‌شود که Display نام دارد. نشانه‌روهای این آرایه، همواره تنظیم می‌شوند که در طی مدت اجرای هر رویه مثل p، به رکوردهای فعالیت رویه‌هایی اشاره کنند که متغیرهای آن‌ها برای رویه p سراسری محسوب می‌شوند. استفاده از Display کار دست‌یابی به متغیرهای سراسری را سرعت می‌بخشد. نحوه کار Display را با استفاده از مثال زیر بررسی می‌کنیم:

```
proc main ( )
   proc p (a)
      proc Q (b)
          call R(x.y)
      end Q
      call Q (z)
   end p;
   Proc R (c.d)
  and R
   call P (w)
end main.
```

هنگامی که رویه main رویه p را فراخوانی می‌کند، محتوای خانه دوم Display یعنی d[2] به رکورد فعالیت رویه p و نشانه‌رو d[1] به رویه main که متغیرهایش برای رویه p سراسری محسوب می‌شوند، اشاره می‌کند (شکل a). هم‌چنین محتوای قبلی d[2] در رکورد فعالیت رویه p ذخیره می‌گردد تا پس از بازگشت از این رویه، جهت اعاده مقدار قبلی d[2] استفاده شود. پس از فراخوانی رویه Q به‌وسیله رویه p، نشانه‌روی d[3] به رکورد فعالیت Q اشاره می‌کند که مقدار قبلی آن در رکورد Q ذخیره می‌گردد (شکل b). d[1] و d[2] که به رکورد فعالیت رویه‌هایی اشاره می‌کنند که برای Q سراسری محسوب می‌شوند؛ بدون تغییر باقی می‌مانند.



پس از فراخوانی رویه R به‌وسیله رویه Q، از آن‌جایی‌که رویه R در سطح دوم است و فقط می‌تواند به متغیرهای خود و برنامه اصلی دسترسی داشته باشد، d[2] به رکورد فعالیت R اشاره می‌کند (شکل c). مقادیر قبلی خانه d[2] که در رکوردهای فعالیت رویه‌های R و Q ذخیره شدند، از طریق یک نشانه‌رو به هم مرتبط هستند. مادامی که کنترل در رویه R است، فقط به دو خانه d[1] و d[2] مراجعه می‌گردد. به عبارت دیگر، به‌طور کلی، اگر کنترل در رویه‌ای در سطح L باشد، فقط می‌توان به خانه‌های d[1] الی d[L] مراجعه کند. پس از خاتمه رویه R و بازگشت به رویه Q وضعیت Display هم به موقعیت قبل از فراخوانی R باز می‌گردد (شکل d).

تجزیه و تحلیل معنایی (Semantic Analysis)

پس از خاتمه مراحل تحلیل واژه‌ای و نحوی، کامپایلر برنامه ورودی را از نظر معنا مورد بررسی قرار می‌دهد. کامپایلرها معمولاً دو نوع بررسی معنایی انجام می‌دهند: بررسی‌های پویا (Dynamic checks) و بررسی‌های ایستا (Static checks)

بررسی‌های پویا، مربوط است به مواردی که کامپایلر اطلاعات کافی برای بررسی آن‌ها ندارد و لذا کدهایی در لابه‌لای کد مقصد قرار می‌دهد تا بررسی‌های لازم در زمان اجرا صورت گیرد. مثلاً اگر در یک برنامه به آرایه‌ای (مثلاً a) با ۱۰ خانه تعریف شده به صورت a[i] رجوع گردد، به‌طوری‌که مقدار اندیس i در زمان کامپایل مشخص نباشد، کامپایلر کدی تولید می‌کند که در زمان اجرا مراقب باشد؛ مقدار اندیس این آرایه بیش‌تر از عدد ۱۰ نگردد.

بررسی‌های معنایی ایستا

بررسی‌های ایستا، مربوط می‌شوند به مواردی که در زمان کامپایل قابل بررسی هستند. بررسی‌های معنایی ایستا را می‌توان به‌صورت زیر دسته‌بندی نمود:

۱ـ بررسی‌های هماهنگی گونه (Type checks) : در این نوع بررسی، یکسان بودن گونه متغیرهای شرکت کرده در عبارت‌ها بررسی می‌شود. مثلاً جمع شدن یک بردار با یک متغیر ساده یک خطای معنایی است و توسط این دسته از بررسی‌های معنایی کشف می‌شود.

۲ـ چک واحد بودن (Uniqueness checks): مواردی در برنامه وجود دارند که باید منحصر به فرد باشند. مثلاً در برخی از زبان‌های برنامه‌سازی، تعریف متغیرها باید منحصر به فرد باشد.

۳ـ بررسی ساختارهای تودرتو ومرتبط به هم (Nested – related checks): در برخی از زبان‌ها، نظیر Ada حلقه‌های تکرار نیز می‌توانند اسم داشته باشند و این اسم باید به‌صورت یکسان هم در ابتدا و هم در انتهای حلقه آورده شود.یا مثلاً بررسی این‌که تعداد پارامترهای واقعی و پارامترهای رسمی یک رویه باید یکسان باشند.

۴ـ بررسی مسیر انتقال کنترل (Control –flow checks): مثلاً دستور break در زبان C باعث می‌شود که از یک block خارج شویم. حال اگر دستور break داخل هیچ بلوکی واقع نباشد، یک خطای معنایی رخ داده است.

جدول علایم (Symbol Table)

کامپایلر برای نگهداری اسامی که برنامه‌نویس انتخاب کرده و اطلاعات مربوط به آن‌ها از جدول علایم استفاده می‌کند. این اطلاعات به‌ویژه در مراحل تحلیل معنایی و تولید کد میانی مورد استفاده قرار می‌گیرد. هر بار که در یک برنامه اسمی مشاهده می‌شود، جدول علایم مورد جستجو قرار گرفته و اگر آن اسم جدید باشد، رکورد جدیدی برایش ایجاد می‌گردد. هر گاه نیز اطلاعات جدیدی در مورد یک اسم یافت شود، رکورد مربوطه در جدول علایم به‌هنگام می‌شود.

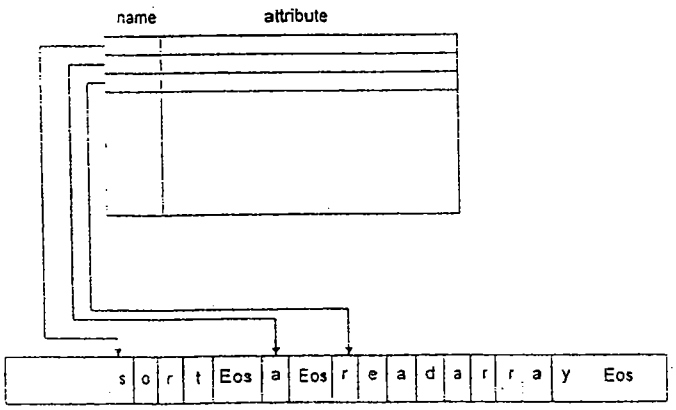
از آن‌جا که مراجعات به جدول علایم وقت بسیار زیادی از کامپایل را به خود اختصاص می‌دهد، ضرورت دارد؛ روش دسترسی به جدول علایم به‌گونه‌ای باشد که این دسترسی راحت‌تر و سریع‌ترانجام شود. معمولاً از یکی از دو روش زیر برای پیاده سازی جدول علایم استفاده می‌شود: روش لیست ترتیبی و روش Hash.

استفاده از لیست ترتیبی، پیاده‌سازی آسان‌تری دارد، لیکن با بزرگ شدن جدول علایم، دست‌یابی به آن کند خواهد شد. از طرف دیگر، استفاده از روش hash که از نظر سرعت بر روش اول، برتری دارد نیز دارای معایبی است: اول، آن‌که یافتن توابعی که بتواند برای اسامی مختلف آدرس‌های مختلف تولید کند مشکل است. ضمناً از نظر پیاده‌سازی نیز مشکل‌تر بوده و حافظه بیش‌تری نیز مصرف می‌کند؛ زیرا امکان دارد اسامی به صورت پراکنده در جدول قرار گیرند. اما در هر حال از روش اول بهتر به نظر می‌رسد.

بهتر است که اندازه جدول علایم به صورت پویا و در حین کامپایل افزایش یابد. در صورتی که اندازه این جدول، ثابت در نظر گرفته شود، باید این اندازه به حد کافی بزرگ باشد که بتواند برای هر برنامه‌های کار کند. هر رکورد در جدول نشانه‌ها، مربوط است به تعریف یک اسم. فرمت این رکورد یکسان نیست؛ زیرا اطلاعاتی که باید در آن‌ها نگهداری شود، بستگی دارد به کاربرد اسم مربوطه که می‌توان به‌صورت زیر، بین این رکوردها نظمی ایجاد کرد:

۱ـ می‌توان به هر نوع از رکوردها یک کد خاص نسبت داد. مثلاً کد ۱ برای رکوردهای مربوط به متغیرهای ساده، کد ۲ برای رکوردهای مربوطه به رویه‌ها و...

۲ـ می‌توان برخی از اطلاعات در مورد اسامی را خارج از جدول نگه داشت و سپس با استفاده از نشانه‌روهایی به آن‌ها دسترسی نمود. به عنوان مثال، در صورتی که بخواهیم از این روش برای اسم متغیرها (Lexemes) استفاده نماییم، می‌توانیم مطابق شکل زیر در فیلد اسامی نشانه‌روهایی به یک آرایه کاراکتری جداگانه به نام جدول رشته‌ها (String Table) قرار می‌دهیم.



نوع رکوردهای جدول علایم، زمانی مشخص می‌شود که وظیفه متغیر معلوم گردد. در مواردی ممکن است یک اسم در برنامه وظایف مختلفی داشته باشد. به عنوان مثال، ممکن است در بخش Declaration زبان C داشته باشیم:

```
int x;
struct x {float , z};
```

در این مثال x هم به عنوان یک متغیر و هم به عنوان یک رکورد با دو فیلد در نظر گرفته شده است. لذا، در جدول علایم، دو رکورد برای x وجود خواهد داشت. در چنین وضعیتی، اسکنر نمی‌تواند که اسم مذکور، یک متغیر است و یا یک رکورد و لذا، پارسر تشخیص نهایی را خواهد داد.

انباره محدود (Scope Stack): کامپایلرها برای تعیین محدوده اعتبار تعریف متغیرها از انباره‌ای به نام انباره محدوده استفاده می‌کنند. هنگامی که متغیری در برنامه اصلی مشاهده شد، خانه‌ای از جدول علایم که مربوط به تعریف آن متغیر است، باید مشخص شود. (در زبان‌های برنامه‌سازی مختلف معمولاً قوانینی وجود دارد که محدوده تعریف متغیرها را مشخص می‌کنند.) یک روش برای تعیین محدوده قانونی متغیرها ایجاد جدول علایم جداگانه برای هر محدوده است. به مثال زیر توجه کنید:

```
proc A
    int x:
    proc B
      real x;
    end B
     proc C
       struct x ;
     end C
end A
```

استفاده از انباره محدوده، به این صورت است که هر گاه ترجمه محدوده جدیدی از متغیرها (مثل یک رویه یا بلوک جدید) آغاز می‌گردد، آدرس شروع جدول علایمی که برای آن محدوده در نظر گرفته شده است، وارد انباره محدوده می‌گردد. هر گاه نیز کار ترجمه یک محدوده خاتمه می‌یابد، آدرس شروع جدول علایم آن محدوده از بالای انباره مزبور، حذف می‌گردد. همان‌گونه که در شکل زیر نشان داده شده است، از آن‌جا که در برنامه فوق متغیر x در سه محدوده مختلف تعریف شده، برای آن سه رکورد مختلف در سه جدول علایم مختلف که هر یک مربوط به یک محدوده متفاوت است، ایجاد می‌شود.

X	Ii	100

جدول علایم A

x	real	200

جدول علایم B

x	struct	300

جدول علایم C

تولید کد میانی (Intermediate Code Generation)

پس از انجام مراحل مختلف تحلیل (واژه‌ای، نحوی و معنایی)، نوبت به تولید برنامه‌های معادل برنامه ورودی به یک زبان میانی می‌رسد. معمولاً از زبانی به عنوان زبان میانی استفاده می‌گردد؛ که هم مستقل از ماشین مقصد باشد و هم بتوان برنامه به زبان میانی را به راحتی بهینه نمود. دستورالعمل‌های زبانی که در این بخش به عنوان زبان میانی مورد استفاده قرار می‌گیرد، دستورات ساده‌ای هستند که به آن‌ها دستورالعمل‌های سه آدرسه (Three Address Codes) گویند. این دستورات، چهارتایی‌های مرتبی هستند که از یک عمل (Operation) و حداکثر سه آدرس تشکیل شده‌اند. مثلاً دستور (:=, A1, A2) محتوای خانه A1 حافظه را به درون آدرس A2 حافظه می‌ریزد. دستورات ساده‌ای نیز برای انتقال کنترل استفاده می‌شود. مثلاً (JPF, A1, A2,) یک دستور، پرش شرطی است و به این صورت، عمل می‌کند که اگر محتوای خانه A1 حافظه نادرست (False) باشد، کنترل به آدرس A2 و در غیر این‌صورت به دستور بعدی منتقل می‌گردد. دستور (JP, A1,,)نیز یک پرش غیرشرطی است که در هر حال کنترل را به خانه A1 منتقل می‌کند.

تولید کد بالا به پایین

تولید کد نیز همانند تحلیل نحوی می‌تواند به دو صورت بالا به پایین و یا پایین به بالا انجام شود. در واقع، جهت تولید کد، تابع جهتی است که پارسر در آن جهت کار می‌کند. یعنی اگر پارسر به‌صورت بالا به پایین کار کند، تولید کد هم به‌صورت بالا به پایین انجام می‌شود. هم‌چنین اگر پارسر به‌صورت پایین به بالا عمل کند، تولید کد نیز پایین به بالا خواهد بود. در این بخش یک روش تولید کد بالا به پایین به همراه روش تجزیه LL(1) معرفی می‌گردد.

لازم به توضیح است که اگرچه مطالب این بخش در مورد تولید کد میانی است، لیکن، کلیه مطالب، برای تولید کد نهایی نیز کاربرد دارد. بنابراین، در طول بخش، از عبارت تولید کد به‌جای عبارت تولید کد میانی استفاده می‌شود. پیش از شروع توضیحات مبحث تولید کد، لازم است که چند مفهوم جدید در ارتباط با این موضوع معرفی گردد.

انباره مفهومی (Semantic Stack): علاوه بر انباره‌های تجزیه (Parse Stack) و محدوده (Scope Stack)، کامپایلرها از انباره دیگری نیز به نام انباره مفهومی (Semantic Stack) که به بخش تولید کد اختصاص دارد، استفاده می‌کنند.

علایم کنش (Action Symbols): علایمی هستند که از آن‌ها در تولید کد بالا به پایین جهت هدایت عمل تولید کد استفاده می‌گردد. این علایم، بایستی در محل‌های مناسبی در سمت راست برخی از قواعد گرامر قرار داده شوند. برای آن‌که علایم کنش از سایر علایم گرامر (پایانه‌ها و غیرپایانه‌ها) تفکیک شوند، معمولاً یک کارکتر ویژه (مثلاً #) در جلوی آن‌ها قرار داده می‌شوند.

کنش‌های مفهومی (Semantic Actions) یا روال‌های مفهومی (Semantic Routines): این‌ها در واقع، زیر روال‌هایی هستند که وظیفه اصلی تولید کد را انجام می‌دهند. به عبارت دیگر، بخش تولید کد یک کامپایلر، تشکیل شده است از یک تعداد زیر رویه موسوم به روال‌های مفهومی . هر روال (یا کنش) مفهومی مرتبط است با یکی از علایم کنش. در طی تجزیه بالا به پایین، هر زمان که یک علامت کنش بالای انباره پارس قرار بگیرد، پارسر برنامه تولید کد را فرا می‌خواند و آن علامت کنش را به عنوان یک پارامتر به برنامه تولید کد ارسال می‌کند. برنامه تولید کد نیز با استفاده از علامت کنش دریافت شده، روال مفهومی مربوطه را پیدا و اجرا می‌کند. اجرای برخی از روال‌های مفهومی سبب تولید کد شده، و اجرای برخی دیگر، تنها انباره مفهومی را به هنگام می‌کند.

بلوک برنامه (Program Block): به بخشی از حافظه زمان اجرا گویند که کد تولید شده توسط کامپایلر در آن قرار گیرد. در این‌جا برای سادگی فرض می‌کنیم، که حافظه زمان اجرا به صورت یک بردار ساده است که در هر خانه آن که با PB[i] نشان داده

می‌شود، یک دستور سه آدرسه و یا یک مقدار قرار می‌گیرد. در ضمن فرض می‌کنیم که کل حافظه زمان اجرا به صورت ایستا تخصیص داده می‌شود. هم‌چنین فرض می‌کنیم، کامپایلر آدرس‌های ۱ الی ۹۹ حافظه را به بلوک برنامه اختصاص می‌دهد.

حافظه داده‌ها (Data Memory): به بخشی از حافظه زمان اجرا گفته می‌شود که به متغیرهای برنامه اختصاص یافته است. در این‌جا برای سادگی فرض می‌کنیم که کامپایلر ما آدرس‌های ۱۰۰ الی ۴۹۹ را به متغیرها و آدرس‌های ۵۰۰ به بالا را به داده‌های موقتی (Temporaries) اختصاص می‌دهد.

تولید کد عبارت جبری و دستور انتساب (Assignment)

حال، به عنوان مثال به گرامر زیر که جملات انتساب (Assignment) را توصیف می‌کند، توجه کنید. در گرامر زیر چهار علامت کنش #assign ،#add ،#mult و #pid وجود دارد که هر یک در محل خاصی از گرامر قرار گرفته‌اند. محل قرار گرفتن این علایم از نظر تولید کد صحیح، اهمیت بسیاری دارد.

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

فرم کلی برنامه تولیدکنندهٔ کد به صورت زیر است که در آن هر یک از بندهای دستور case یک روال مفهومی را تشکیل می‌دهند:

```
procedure generate (action)
  begin
    case (action)
        #assign    begin...end
        #pid:      begin...end
        #add:      begin...end
        #mult:     begin...end
        ...
      end
end
```

روال #pid (که آن‌را push id می‌خوانیم)، آدرس تخصیص یافته به توکن جاری را که در جدول علایم درج شده، به‌وسیله اجرای تابعی بنام Findaddr یافته و وارد انباره مفهومی می‌کند. این رویه به صورت زیر نوشته می‌شود. توجه داشته باشید که کلیه دستورات push و pop در روال‌های مفهومی مربوط به انباره مفهومی است.

```
#pid:begin
       p ← Findaddr(input)
       push (p)
 end
```

روال #add که در ادامه آورده شده است، ابتدا تابعی به‌نام gettemp را که آدرس اولین خانه آزاد حافظه موقتی را بر می‌گرداند، فراخوانی می‌کند. سپس دستور سه آدرسه انجام عمل جمع را تولید می‌کند؛ یعنی آن را در خانه PB[i] قرار می دهد. ما فرض می‌کنیم که یک شمارنده سراسری i وجود که به خانه‌های بلوک برنامه اشاره می‌کند. در شروع کار تولید کد، این شمارنده به اولین خانه خالی بلوک برنامه اشاره می‌کند. روال‌های مفهومی باید طوری مقدار این شمارنده را کنترل کنند، که همواره به اولین خانه خالی

PB اشاره کند. عملوندهای عمل جمع قبل از آن‌که نوبت به اجرای روال #add برسد، توسط روال #pid وارد انباره مفهومی می‌شوند. حاصل جمع نیز در یک حافظه موقتی به آدرس t که از تابع gettemp گرفته شده قرار می‌گیرد.

```
#add : begin
    t ← gettemp;
    PB[i] ← (+,ss(top),ss(top−1),t)
    i ← i+1
    pop(2)
    push(t)
end
```

روال #mult با یک اختلاف جزیی در ارتباط با عمل دستور سه آدرسه تولید شده عیناً مطابق روال #add است:

```
#mult:begin
    t ← gettemp;
    PB[i] ← (*,ss(top),ss(top−1),t)
    i ← i+1
    pop(2)
    push(t)
end
```

روال #assign کد سه آدرس‌های برای کپی کردن یک خانه از حافظه در یک خانه دیگر تولید می‌کند.

```
#assign : begin
    PB[i] ← (:=,ss(top),ss(top−1),)
    i ← i+1
    pop(2)
end
```

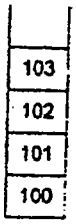
شکل زیر جدول LL(1) گرامر فوق را نشان می‌دهد:

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

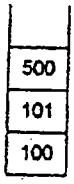
می‌خواهیم برای عبارت a:=b+c*d کد تولید کنیم. فرض کنید آدرس‌های ۱۰۰ الی ۱۰۳ به ترتیب به متغیرهای a,b,c و d اختصاص یافته است. جدول زیر تجزیه قدم به قدم این رشته و ترتیب اجرای روتین‌های مفهومی را نشان می‌دهد. برای خوانایی بیش‌تر، توکن متغیرها به صورت id_j نمایش داده شده که در آن اندیس j ترتیب دیده شدن متغیر در ورودی را نشان می‌دهد.

محتوای انباره	ورودی
$ S	$id_1 := id_2 + id_3 * id_4$
$#assign E := L	$id_1 := id_2 + id_3 * id_4$
$#assign E := id#pid	$id_1 := id_2 + id_3 * id_4$
$#assign E′T	$id_2 + id_3 * id_4$
$#assign E′T′F	$id_2 + id_3 * id_4$
$#assign E′T′id#pid	$id_2 + id_3 * id_4$
$#assign E′T′	$+id_3 * id_4$
$#assign E′	$+id_3 * id_4$
$#assign E′#add T +	$+id_3 * id_4$
$#assign E′#add T	$id_3 * id_4$
$#assign E′#add T′F	$id_3 * id_4$
$#assign E′#add T′id#pid	$id_3 * id_4$
$#assign E′#add T′	$*id_4$
$#assign E′#add T′#multF*	$*id_4$
$#assign E′#add T′#mult F	id_4
$#assign E′#add T′#mult id#pid	id_4
$#assign E′#add T′#mult	$
$#assign E′#add T′	$
$#assign E′#add	$
$#assign E′	$
$#assign	$
$	$

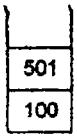
ترتیب اجرای روتین‌های مفهومی، به این صورت است که ابتدا، چهار مرتبه متوالیاً روال pid# اجرا شده، و به دنبال آن روال‌های mult# , add# و assign# به ترتیب اجرا می‌شوند. اشکال زیر وضعیت انباره مفهومی را در قدم‌های مختلف کامپایل نشان می‌دهد.



وضعیت semantic-stack
بعد از انجام * روتین PID#

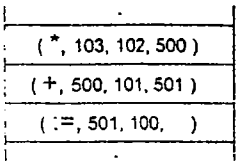


وضعیت ss
بعد از انجام عمل mult#



وضعیت ss
بعد از انجام عمل ADD#

بعد از انجام روتین assign# انباره مفهومی خالی خواهد شد. شکل زیر کد تولید شده برای جمله انتساب فوق را نشان می‌دهد. همان‌گونه که مشاهده می‌شود، تقدم عملیات در این کد رعایت شده است.

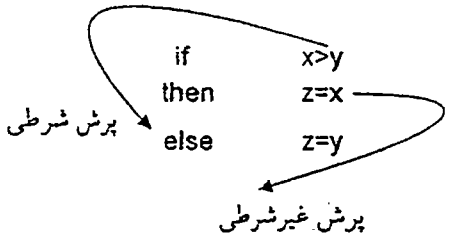


کد تولید شده برای عبارت مورد نظر

تولید کد جملات شرطی

در ادامه، نحوه تولید کد جملات شرطی توضیح داده می‌شود. قاعده زیر که در آن غیر پایانه BE معرف یک عبارت منطقی است، جملات شرطی دارای جمله else را توصیف می‌کند. هم‌چنین شکل زیر با توجه به مفهوم (Semantic) جملات شرطی، عملیاتی را که در حین اجرای چنین جملاتی باید انجام شود، نشان می‌دهد. به عنوان نمونه، در جمله زیر پس از مشخص شدن درست یا نادرست بودن شرط جمله، باید یک پرش شرطی به ابتدای قسمت else جمله انجام شود. در انتهای قسمت then نیز باید یک پرش بدون شرط از روی قسمت else وجود داشته باشد تا اگر چنان‌چه شرط برقرار بود و قسمت then اجرا گردید، دیگر قسمت else اجرا نشود.

$ST \rightarrow$ if BE then ST else ST



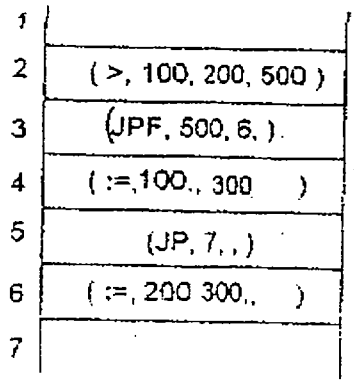
پس از تولید کد برای شرط x>y باید کدی برای پرش شرطی تولید شود، ولی مشکل آن است که آدرس محلی که باید پرش به آن صورت گیرد، هنوز مشخص نیست. برای رفع این مشکل یک خانه خالی برای پرش شرطی رزرو می‌شود و هنگامی که آدرس موردنظر (یعنی آدرس اولین دستور بخش else) مشخص شد به عقب باز می‌گردیم و کد پرش شرطی را در خانه رزرو شده، قرار می‌دهیم. به این عمل اصطلاحاً Backpatching گویند. در مورد پرش غیرشرطی نیز همین مشکل را داریم. در قاعده زیر، محل درست علایم کنش لازم برای تولید کد جملات شرطی قرار داده شده است.

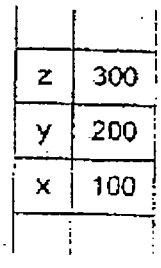
$ST \rightarrow$ if BE #save then ST #jpf – save else ST # jp

روال‌های مفهومی مرتبط با علایم کنش فوق به صورت زیر است:

```
#save:=begin
        push(i)
        i → i+1
end
#jpf-save:begin
        PB[ss(top)] ← (JPF,ss(top – 1),i + 1,)
        Pop(2)
        Push(i)
        i → i+1
end
#jp:begin
      PB[ss(top)] ← (JP,i,,)
      pop(1)
end
```

شکل زیر جدول علایم و کد تولید شده برای جمله شرطی فوق را نشان می‌دهد.





جدول علائم

حال اگر بتوانیم جملات شرطی بدون else نیز داشته باشیم. باید از قواعد و کنش‌های به فرم زیر استفاده کنیم.

$ST \rightarrow if\ \ BE\ \#\ save\ Then\ ST\ ELSE$

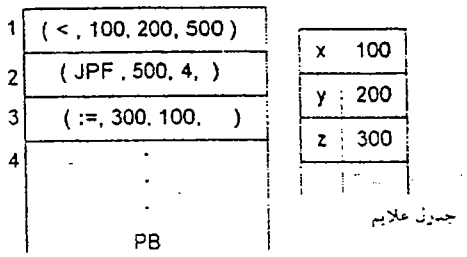
$ELSE\ \rightarrow \varepsilon \#jpf$

$ELSE \rightarrow else \#\ jpf - save\ \ \ ST \#\ jp$

در این حالت روال جدیدی به نام مثلاً #jpf و به شکل زیر نیاز داریم.

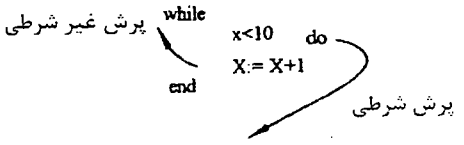
```
#jpf:begin
        PB[ss(top)] ← (JPF, ss(top−1), i,)
        pop(2)
end
```

فرض کنید می‌خواهیم برای جمله if x<y then z:=x کد تولید کنیم. شکل زیر، جدول علایم فرض شده و کد تولید شده توسط روال فوق را نشان می‌دهد.



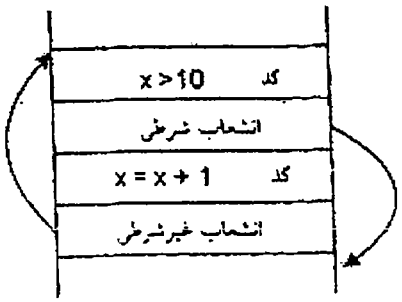
تولید کد حلقه‌های تکرار

در ادامه، علایم کنش لازم و روال‌های مربوطه برای تولید کد چند فرم حلقه تکرار آمده است. در هر مورد با استفاده از روال‌های نوشته شده برای یک مثال کوچک نیز کد سه آدرسه تولید شده است. (به عنوان یک تمرین می‌توانید درستی و روال‌ها و تولید کد را به صورت قدم‌به‌قدم وارسی کنید.) ابتدا به روال‌های تولید کد حلقه‌های تکرار While می‌پردازیم.



ST → while #label BE do #save ST #while end

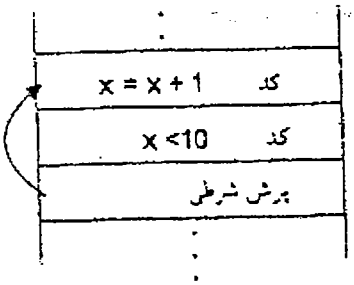
```
#label :begin
    push(i)
end
#while : begin
    PB [ss(top)] ← (JPF, ss(top − 1), i + 1,)
    PB[i] ← (JP. ss(top − 2),,)
        pop(3)
        i ← i + 1
end
```



مثال بعدی، حلقه‌های تکرار do-until است که در آن‌ها قبل از بررسی شرط حلقه، بدنه حلقه یک‌بار اجرا می‌گردد:

```
do
x=x+1
until x<10  end
```

ST → do #lable ST until BE #until end




```
#until : begin
    PB[i] ← (JPF, ss(top).ss(top − 1),)
    pop(2)
    i ← i + 1
end
```

مثال و روال‌های مفهومی بعدی مربوط به حلقه‌های تکرار For است. مثال در مورد حالتی است که step نداریم:

```
for j=1 to 9  Do
    x=x+j
end
ST → for # pid # pid id := E # assign  to  E  # comp-save  STEP do ST  end  # for
STEP → ε # step  | by  E
#comp-save:begin
    t ← gettemp
    PB[i] ← (<, ss(top-1), ss(top), t)
    i ← i+1
    pop(1)
    push(t)
    push(i)
    i ← i+1
end
#for:begin
    PB[i] ← (+, ss(top), ss(top-3), ss(top-3))
    i ← i+1
    PB[i] ← (JP, ss(top-1)-1, , )
    i ← i+1
    PB[ss(top-1)] ← (JPT, ss(top-2), i)
    pop(4)
end
#step :begin
    t ← gettemp
    PB[i] ← (:=, #1, t, )
    i ← i+1
    push(t)
end
```

تولید کد دستور goto

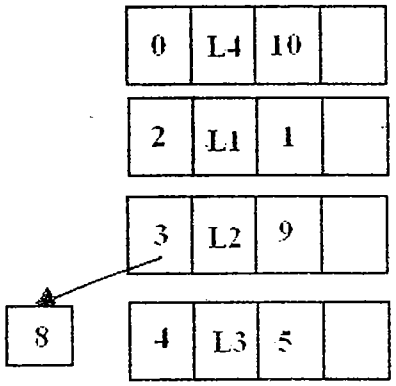
تولید کد جملات goto رو به جلو نیاز به اجرای اعمال خاصی دارد. تولید کد دستور goto رو، به عقب بسیار ساده است. در این حالت، چون قبل از رسیدن به دستور مزبور برچسب مربوطه دیده شده و آدرس آن مشخص است، یک پرش بدون شرط به آن آدرس تولید می‌گردد. در مورد دستورات goto رو به جلو، چون در هنگام رسیدن به دستور، هنوز آدرس برچسب مربوطه مشخص نیست، کد پرش بدون شرط را نمی‌توان ایجاد کرد. در این مورد، بایستی یک خانه خالی رزرو نمود و آدرس آن خانه خالی را در انباره مفهومی ذخیره نمود، تا پس از رسیدن به برچسب مربوطه با استفاده از عمل Backpatching خانه خالی رزرو شده با کد پرش غیر شرطی تکمیل گردد. لیکن، از آن‌جا که در حالت کلی، می‌توان چندین دستور goto رو به جلو، به یک برچسب مشترک داشت، در هنگام رسیدن به یک برچسب مشخص نیست که دقیقاً چه تعداد از آدرس‌های ذخیره شده در انباره مفهومی مربوط به دستورات goto رو به جلو است. یکی از روش‌های حل این مشکل، ایجاد یک لیست پیوندی اختصاصی برای ذخیره اطلاعات مربوط به برچسب‌ها و دستورات goto است. هر گره از این لیست دارای چهار فیلد اطلاعاتی به‌صورت زیر است:

نشانه‌رو به گره دیگر	آدرس برچسب	برچسب	آدرس goto مربوط به برچسب

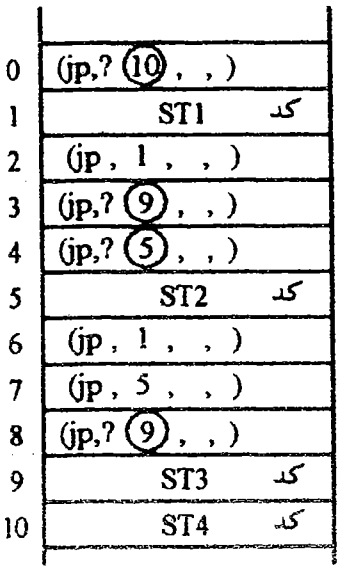
حال به عنوان مثال برنامه زیر را در نظر بگیرید:

```
    goto L4
L1:ST1
    goto L1
    gotoL2
    gotoL3
L3:ST2
    goto L1
    goto L3
    goto L2
L2:ST3
L4:ST4
```

برای مثال فوق، لیست پیوندی مزبور در پایان کار تولید به صورت زیر خواهد بود. در مواردی که بیش‌تر از یک دستور goto رو به جلو به یک برچسب مشترک موجود داشته باشد (مثلاً goto L2)، آدرس goto های دوم به بعد با استفاده از نشانه روهای اضافی به گره مربوطه متصل می‌گردد.



شکل بعد، کد تولید شده برای مثال فوق را نشان می‌دهد. در محل‌هایی که کنار آدرس پرش علامت "?" قرار داده شده است. آدرس مزبور با استفاده از عمل Backpatching تکمیل شده است.



تولید کد پایین به بالا

تولید کد پایین به بالا، هنگامی موردنیاز است که تحلیل نحوی به‌صورت پایین به بالا اجرا گردد. همان گونه که ملاحظه شد، یک پارسر LL(1) هر گاه، یک علامت کنش بالای انباره تجزیه قرار گیرد، رویه تولید کد را فرا می‌خواند. حال آن‌که پارسرهای پایین به بالا هر گاه عمل کاهش صورت پذیرد، رویه تولید کد را فرا می‌خوانند. در روش تولید کد پایین به بالا علامت کنشی نداریم و تنها هنگامی فراخوانی رویه تولید کد، شماره قاعده‌ای که در انجام کاهش شرکت داشته است به عنوان پارامتر به رویه تولید کد منتقل می‌شود. این درست به مثابه آن است که تمام علایم کنش، در انتهای سمت راست قواعد قرار گرفته باشند. به همین دلیل، اگر در قاعده‌ای که برای تولید کد بالا به پایین طراحی شده، علایم کنشی درمحلی به غیر از انتهای سمت راست قاعده قرار داشته باشند، آن قاعده قابل استفاده در تولید کد پایین به بالا نیست. در این شرایط باید قواعد را به‌گونه‌ای تغییر داد که بدون آن‌که زبان گرامر و یا ترتیب منطقی اجرای روال‌های مفهومی تغییر کند، کلیه علایم کنش در انتهای سمت راست قواعد قرار گیرند. به عنوان یک نمونه مجدداً به گرامر معرف حلقه‌های تکرار for و علایم کنش لازم برای تولید کد این‌گونه حلقه‌ها توجه کنید.

$ST \rightarrow$ for #pid #pid id := E #assign to E #comp – save STEP do ST end #for

$STEP \rightarrow \varepsilon$ #step | by E

قواعد فوق برای استفاده در تولید کد پایین به بالا بایستی به صورت زیر تغییر کنند.

$ST \rightarrow$ FOR – HEAD STEP do ST end #for

FOR – HEAD $\rightarrow$ FROM – PART to E #comp – save

FROM PART $\rightarrow$ FOR ID := E #assign

For – ID $\rightarrow$ for id #pid # pid

$STEP \rightarrow \varepsilon$ #step | by E

اگر روش تجزیه پایین به بالا اجازه استفاده از قواعد اپسیلون را بدهد، می‌توان گرامر فوق را به صورت زیر نیز تغییر داد که در آن A_1، A_2 و A_3 غیرپایانه‌های جدید هستند. این روش تغییر ضمن سادگی بیش‌تر به خوانایی گرامر نیز کمتر لطمه می‌زند.

$ST \rightarrow$ for A_1 id := E A_2 to E A_3 STEP do ST end #for

$STEP \rightarrow \varepsilon$ #step | by E

$A_1 \rightarrow \varepsilon$ #pid #pid

$A_2 \rightarrow \varepsilon$ #assign

$A_3 \rightarrow \varepsilon$ #comp – save

بهینه‌سازی (Optimization)

مرحله بهینه‌سازی کامپایلر، در صورت وجود، از وقت‌گیرترین مراحل ترجمه است. هم‌چنین اگر بخواهیم کامپایلر بهینه‌سازی نیز انجام دهد، دیگر نمی‌توان آن‌را به‌صورت تک گذره (One Pass) پیاده‌سازی نمود. بهینه‌سازی، منطقاً باید روی بخش‌هایی از برنامه ورودی تمرکز کند که قرار است، زیادتر از نقاط دیگر اجرا می‌شوند به همین دلیل، حلقه‌های تکرار محل مناسبی برای تمرکز عملیات بهینه‌سازی هستند.

به‌طور کلی در عمل بهینه‌سازی برنامه ورودی به برنامه معادل دیگری تبدیل (Transform) می‌شود. خصوصیاتی که تبدیلات یک بهینه‌ساز باید داشته باشد عبارتند از:

۱ـ تبدیل، نباید معنی یک برنامه را عوض کند. یعنی اگر برنامه، قبل از بهینه‌سازی فاقد خطا بوده است، انجام بهینه‌سازی نبایستی خطای جدیدی در برنامه ایجاد کند.

۲ـ یک تبدیل باید به‌طور متوسط، سرعت برنامه‌ها را تا حد قابل محاسبه‌ای بالا ببرد. مثلاً اگر وقت بسیار زیادی صرف حذف یک دستور ساده عمل جمع که قرار است در زمان اجرا تنها یک مرتبه اجرا گردد، سرعت برنامه را آن‌قدر ناچیز، بهبود می‌دهد که حتی قابل اندازه‌گیری نیست و لذا ارزش وقت صرف شده را ندارد.

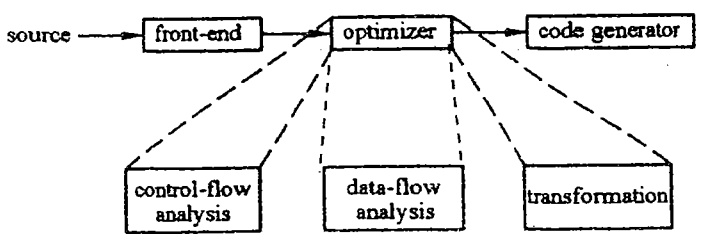
۳ـ یک تبدیل باید ارزش کوششی را که صرف اجرای آن می‌شود، داشته باشد. به عنوان نمونه، برنامه‌های پروژه‌های دانشجویی که معمولاً لازم است بارها کامپایل شده، لیکن دفعات زیادی اجرا نخواهند شد، نیازی به بهینه‌سازی ندارند.

پیش از این‌که بهینه‌ساز، تبدیلی انجام دهد، دو مرحله تحلیلی دیگر باید روی برنامه ورودی که به فرم میانی تبدیل شده است، صورت پذیرد. این مراحل عبارتند از:

۱ـ تحلیل جریان کنترل (Control Flow Analysis)، که در آن برنامه ورودی به یک سری بلوک‌های پایه (Basic Blocks) مجزا تبدیل می‌شود. هر بلوک پایه تشکیل شده است از تعدادی دستورالعمل ساده غیر پرشی که به‌صورت ترتیبی اجرا می‌شوند. آخرین دستورالعمل هر بلوک پایه، یک دستور پرشی است که کنترل را به یک بلوک پایه دیگر منتقل می‌کند.

۲ـ تحلیل جریان داده‌ها (Data Flow Analysis)، که تعیین می‌کند در چه محدوده‌ای از برنامه، مقدار متغیرها ثابت باقی مانده و در چه محل‌هایی مقدار جدیدی به‌خود می‌گیرند.

شکل زیر ساختار کلی یک بهینه ساز را نشان می‌دهد:



تبدیلات بهینه‌سازی

همان‌گونه که در بالا اشاره گردید، از مهم‌ترین ویژگی‌هایی که تبدیلات یک بهینه‌ساز باید داشته باشد، عدم تغییر معنی برنامه ورودی است. در ادامه، تعدادی از تبدیلات متداول که معنای برنامه ورودی را حفظ می‌کنند، معرفی می‌گردد.

۱ـ حذف زیر عبارت مشترک (Common Subexpressions Elimination): عبارتی مانند E را یک زیر عبارت مشترک نامند، هر گاه، E در برنامه قبلاً محاسبه شده باشد و مقدار متغیرهایش در فاصله دو محاسبه تغییری نکرده باشند. در این صورت از محاسبه مجدد E می‌توان خودداری کرد.

۲ـ انتشار کپی (Copy Propagation): در این تبدیل، اگر در یک بلوک پایه دستوری به فرم x:=y (که به آن دستور کپی گفته می‌شود) داشته باشیم، در دستورات بعدی تا جایی که مقدار متغیرهای x و y تغییر نکرده‌اند، هر کجا که از متغیر x استفاده شده است، آن‌را با متغیر y جایگزین می‌کنیم. این عمل، به‌خودی خود برنامه را بهبود نمی‌دهد، بلکه زمینه را برای اجرای تبدیل بهینه‌سازی بعدی (حذف کد مرده) فراهم می‌کند.

۳ـ حذف کد مرده یا کد غیر قابل دسترس (Unreachable and Dead Code Elimination): کد مرده، به دستوراتی گفته می‌شود که مقادیری را محاسبه می‌کنند، لیکن از آن‌ها هرگز استفاده نمی‌شود. برخی اوقات نیز شرایط دستورات شرطی به‌گونه‌ای است که در زمان کامپایل مشخص می‌گردد که به بخشی از کد، هرگز مراجعه نخواهد شد؛ به آن بخش، کد غیرقابل دسترس گویند. در این تبدیل، کدهای مرده و غیرقابل دسترس حذف می‌شوند.

۴ـ جاگذاری مقادیر ثابت (Constant Folding): در مواردی که مقدار یک عبارت در زمان کامپایل قابل محاسبه است و برابر با یک مقدار ثابت می‌شود، این تبدیل، مقدار عبارت را محاسبه و جای‌گذاری می‌کند.

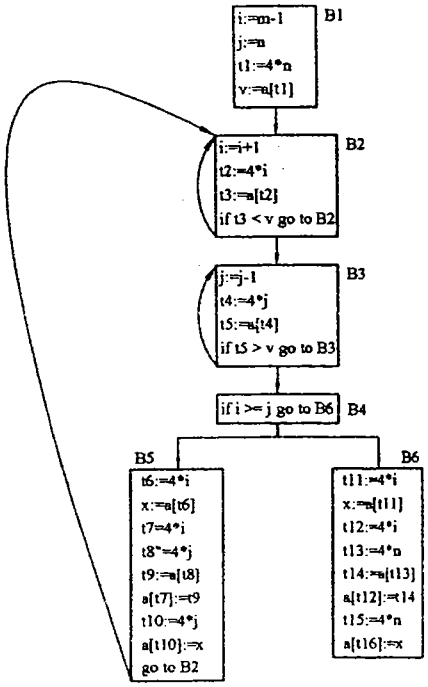
حال با استفاده از برنامه quicksort زیر به بررسی تبدیلات بهینه‌سازی فوق می‌پردازیم:

```
void quicksort (m,n)
int m,n,
{
int  i,   j;
int  v,   x;
if  (n<=m) return;
/*fragment begins here */
i=m-1;j=n;   v=a[n];
while (1) {
do  i=i+1; while (a[i]<v);
do  j=j-1; while (a[j]>v);
if (i > = j) break;
x = a [i] ; a [i] = a[j]; a [j] = x ;
}
x = a [i] ; a [i] = a [n] ; a [n] = x ;
/* fragment ends here */
quicksort (m,  j) ;
quicksort (I + 1 , n );
}
```

کد سه آدرسه تولید شده برای برنامه فوق به صورت زیر خواهد بود:

(1)	$i := m-1$	(16)	$t_7 := 4*i$
(2)	$j := n$	(17)	$t_8 := 4*j$
(3)	$t_1 := 4*n$	(18)	$t_9 := a[t_8]$
(4)	$v := a[t_1]$	(19)	$a[t_7] := t_9$
(5)	$i := i+1$	(20)	$t_{10} := 4*j$
(6)	$t_2 := 4*i$	(21)	$a[t_{10}] := x$
(7)	$t_3 := a[t_2]$	(22)	goto(5)
(8)	if $t_3 < v$ goto (5)	(23)	$t_{11} := 4*i$
(9)	$j := j-1$	(24)	$x := a[t_{11}]$
(10)	$t_4 := 4*j$	(25)	$t_{12} := 4*i$
(11)	$t_5 := a[t_4]$	(26)	$t_{13} := 4*n$
(12)	if $t_5 > v$ goto(9)	(27)	$t_{14} := a[t_{13}]$
(13)	if $i >= j$ goto(23)	(28)	$a[t_{12}] = t_{14}$
(14)	$t_6 := 4*i$	(29	$t_{15} := 4*n$
(15)	$X := a[t_6]$	(30)	$a[t_{15}] := x$

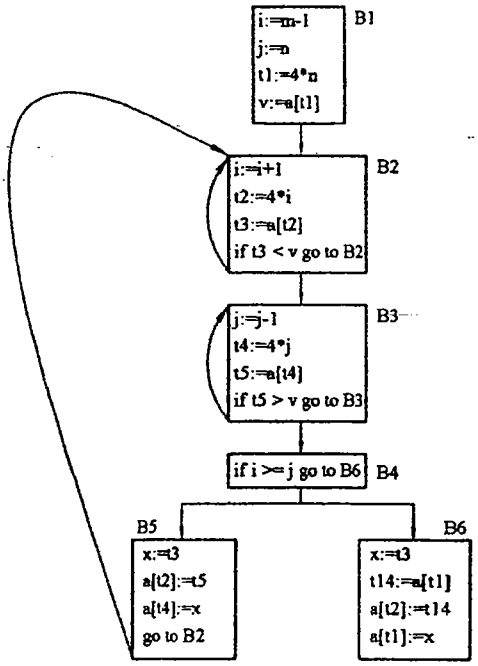
پس از اجرای تحلیل جریان کنترل، این کدها به صورت ۶ بلوک پایه B1 الی B6 تبدیل می‌شوند. در شکل زیر، این بلوک‌ها نشان داده شده‌اند. حال در ادامه، نحوه و اثر هر یک از تبدیلات فوق را بر روی این بلوک‌ها بررسی می‌کنیم.



بلوک B5 نمودار فوق را در نظر بگیرید. از روی تحلیل جریان داده‌ها می‌توان فهمید که مقدار متغیرهای t_7 و t_{10} به ترتیب شامل زیر عبارت‌های مشترک 4*i و 4*j می‌باشند و می‌توان آن‌ها را حذف کرد. در این صورت بلوک B5 به‌صورت زیر تبدیل خواهد شد.

$t_6 := 4*i$
$x := a[t_6]$
$t_8 := 4*j$
$t_9 := a[t_8]$
$a[t_6] := t_9$
$a[t_8] := x$
goto B2

در صورتی که زیر عبارات مشترک در یک بلوک باشند، عمل را حذف زیر عبارات مشترک محلی گویند؛ مانند انچه که در B5 انجام شد. اگر زیر عبارات مشترک در بلوک‌های متفاوتی باشند، عمل حذف را سراسری گویند، مانند کد t2:=4*i در b2 و کد t6 := 4*i در B5 که باعث حذف کد $t6 = 4*i$ می‌گردد. شکل زیر نمودار بلوکی کدهای برنامه quicksort را بعد از حذف کلیه زیر عبارت‌های مشترک محلی و سراسری نشان می‌دهد.



در نمودار فوق هنوز با استفاده از تبدیلات انتشار کپی و حذف کد مرده، می‌توان بهینه‌سازی بیش‌تری انجام داد. بلوک B5 بعد ازعمل انتشار کپی به‌صورت زیر در خواهد آمد:

$x := t_3$
$a[t_2] := t_5$
$a[t_4] := t_3$
goto B_2

با اجرای تبدیل فوق، کد x=t3 به‌صورت کد مرده تبدیل شده و می‌توان آن را حذف کرد.

به عنوان مثالی از حذف کد غیر قابل دسترس به قطعه برنامه زیر توجه کنید:

```
debug=false
....
If (debug) then print...
```

اگر مقدار متغیر debug تا رسیدن به جمله شرطی تغییر نکرده باشد، دستور print برنامه فوق، هرگز اجرا نمی‌شود و در واقع یک کد غیر قابل دسترس است و می‌توان آن را حذف نمود.

مثال زیر نمونه‌ای از اجرای تبدیل جاگذاری مقادیر ثابت است.

```
c = 1
for  i = 1 to 100
   e = 2 + 7 + c
end
```

در این‌جا، با جای‌گذاری مقدار به جای c و محاسبه و جای‌گذاری مقدار 9 به جای 7+2 دستور داخل حلقه به‌صورت e=10 تبدیل می‌شود.

بهینه‌سازی حلقه‌ها

همان‌گونه که در ابتدای این فصل اشاره گردید، مناسب‌ترین محل برای تمرکز عملیات بهینه‌سازی حلقه‌های تکرار هستند. سه تبدیل زیر ویژه بهینه‌سازی حلقه‌های تکرار است.

۱ـ انتقال کد (Code Motion)

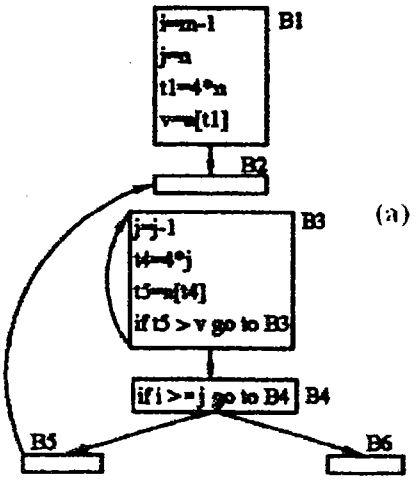
۲ـ حذف متغیرهای استقرایی (Induction Variables Elimination)

۳ـ کاهش پیچیدگی (Reduction in Strength)

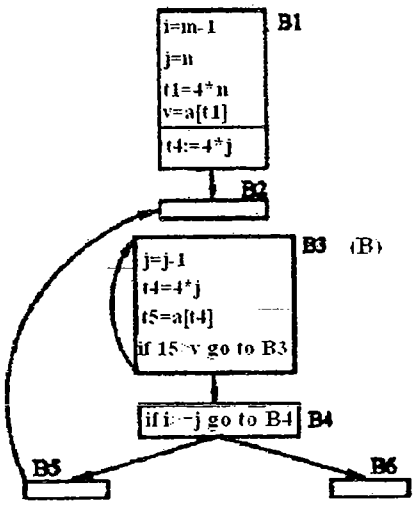
هر گاه، دستوری که در یک حلقه، عمل ثابتی را انجام دهد که تغییر خاصی در روند اجرای حلقه نداشته باشد، آن دستور به خارج حلقه منتقل می‌شود. به این عمل جابه‌جایی کدگویند. به عنوان نمونه دستور e=.... در مثال فوق را می‌توان به خارج حلقه منتقل نمود. به عنوان یک نمونه دیگر به برنامه زیر توجه کنید. در این برنامه محاسبه 2+C به صورت تکراری داخل حلقه انجام می‌شود و بهتر است به بیرون حلقه انتقال یابد.

```
While (limit <2+c) do
    ...
end
```

اگر در یک حلقه مقدار متغیری به‌طور خطی، افزیش و یا کاهش یابد، به آن متغیر استقرایی گویند. مثلاً در بلوک B2 نمودار برنامه quicksort در بالا، متغیرهای i و t2 متغیرهایی استقرایی می‌باشند. اگر حلقه‌ای، n متغیر استقرایی داشته باشد، ممکن است بتوان همه را (بغیر از یکی) حذف و به‌جای آن‌ها از تنها متغیر استقرایی باقی‌مانده استفاده کرد. به این ترتیب کدهایی که در آن‌ها به متغیرهای دیگر مقدار می‌دهیم به عنوان کد مرده شناخته شده و می‌توان آن‌ها را حذف کرد. در این جهت، حتی اگر کاهش دستورات داخل حلقه. باعث افزایش دستورات خارج حلقه شود، اشکالی ندارد؛ زیرا کد خارج از حلقه نسبت به کد داخل حلقه، دفعات کمتری اجرا خواهد شد.



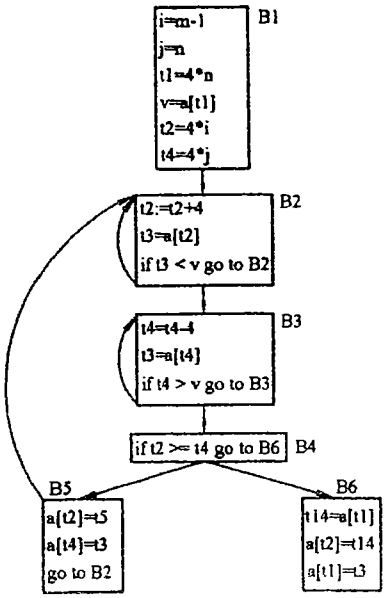
(a)



آخرین تبدیل ویژه حلقه‌ها، کاهش پیچیدگی است. در این تبدیل، سعی می‌شود اعمال پیچیده‌تر با عملیات ساده‌تر جایگزین گردند. مثلاً در صورت امکان یک عمل ضرب به عمل ساده‌تر جمع تبدیل شود. البته این تبدیل اگر در خارج حلقه‌ها صورت گیرد، ارزش چندانی ندارد.

حال به عنوان یک مثال از تبدیل فوق، به قسمت (a) شکل فوق که بخشی از نمودار برنامه quicksort را نشان می‌دهد، توجه کنید. با توجه به کدهای j=j-1 و t4=4*j می‌توان با استفاده از تبدیل کاهش پیچیدگی کد t4=t4-4 را به جای t4=4*j قرار داد. زیرا هر بار یک واحد از مقدار متغیر j و در نتیجه چهار واحد از مقدار متغیر t4 کاسته می‌شود. تنها مسأله این‌جاست که با وارد شدن به بلوک B3 متغیر t4 مقدار ندارد. بنابراین، می‌توان عبارت t4=4*j را در خارج از بلوک B3 قرار داد. نتیجه این کار به صورت قسمت (b) در شکل قبلی خواهد بود:

همان‌طوری که گفته شد متغیرهای i و t2 در بلوک B2 و متغیرهای j و t4 در بلوک B3 متغیرهای استقرایی هستند. با ایجاد تغییراتی در برنامه، می‌توان عمل حذف متغیرهایی استقرایی را نیز در این برنامه اعمال نمود، که به این ترتیب نمودار زیر حاصل خواهد شد.



بهینه‌سازی کد نهایی

برخلاف بهینه‌سازی کد بینابینی، که معمولاً بخش‌های وسیعی از برنامه، مورد تحلیل قرار می‌گیرد، در بهینه‌سازی کد نهایی، اغلب، بخش بسیار کوچکی از برنامه مورد بررسی قرار می‌گیرد. از این رو به بهینه‌سازی کد مقصد، Peephole Optimization گویند. برنامه‌های وجود ندارد که بتواند کد بینابینی تولید شده را به کد نهایی بهینه (optimum) تبدیل کند. با این وجود، می‌توان ملاحظاتی را در نظر گرفت که به‌وسیله آن‌ها کد نهایی بهتری تولید خواهد شد. موارد زیر از جمله این ملاحظات است:

حذف load و store های بی‌فایده، و نگهداری مقدار متغیرها در ثبات‌ها

حذف load و store های پیاپی

استفاده کمتر از ثبات‌ها با تغییر شکل دادن فرم عبارت

استفاده از انتقال به چپ و راست به‌جای ضرب و تقسیم بر توان‌های ۲

سؤال‌ها

آزمون کارشناسی ارشد سال ۱۳۸۲

۱ ـ گرامر SLR(1) دستورات شرطی در یک زبان برنامه‌سازی به شرح مقابل مفروض است در این زبان هر دستور if با یک واژه (پایانه) به نام endif تمام می‌شود. BE مواد عبارات منطقی و other نشانه سایر دستورات (غیر از دستورات شرطی) است. برای این زبان دو کامپایلر یکی با روش تجزیه LL(1) و دیگری با روش تجزیه SLR(1) مفروض است. حداقل تعداد قواعد تولید لازم به منظور تولید کد دستورات شرطی (بدون احتساب قاعده شماره ۳) در دو روش مزبور به ترتیب کدام است؟

1) ST →
if BE then ST endif
2) ST →
if BE then ST else ST endif
3) ST → other

۱) 2 و 2 ۲) 3 و 2 ۳) 3 و 4 ۴) 3 و 6

۲ ـ اگر A یک متغیر از نوع آرایه یک بعدی با اندیس‌های 1 تا 10 و I یک عدد صحیح باشد، خطای اندیس خارج از محدوده (subscript out of range) که در دستورات I := 11 ; B := A[I] وجود دارد، در حالت کلی در چه زمانی و توسط چه برنامه‌های قابل کشف است؟

۱) زمان اجرا توسط سیستم عامل
۲) زمان کامپایل توسط تجزیه کننده دستوری (Parser)
۳) زمان کامپایل توسط تجزیه کننده مفهومی (Semantic Analyzer)
۴) زمان اجرا توسط خود برنامه حاوی دستورات فوق

۳ ـ کدام‌یک از گزینه‌های در خصوص نوع گرامر مقابل صحیح است؟ (λ معرف رشته به طول صفر است)

$S \rightarrow aACb$
$A \rightarrow b / \lambda$
$C \rightarrow cC / \lambda$

۱) گرامر LL(1) نیست، SLR(1) هم نیست.
۲) گرامر LL(1) نیست، اما SLR(1) هست.
۳) گرامر LL(1) است، SLR(1) هم هست.
۴) گرامر SLR(1) نیست، LL(1) هست.

۴ ـ گرامر عبارت جبری مقابل را در نظر بگیرید. درجدول تجزیه SLR (1) گرامر داده شده چه تعداد دستور انقباض یا کاهش (Reduce) با قاعده $F \rightarrow (E)$ وجود دارد؟ (در جدول تجزیه علاوه بر واژه‌های زبان (پایانه‌ها)، ستونی نیز برای واژه $ (مشخص‌کننده انتهای ورودی) در نظر بگیرید. λ معرف رشته به طول صفر است)

$E \rightarrow TE'$
$E \rightarrow -TE'$
$E' \rightarrow \lambda$
$E' \rightarrow +TE'$
$E' \rightarrow -TE'$
$T \rightarrow FT'$
$T' \rightarrow \lambda$
$T' \rightarrow *FT'$
$T' \rightarrow /FT'$
$F \rightarrow id$
$F \rightarrow (E)$

۱) 5 ۲) 6 ۳) 7 ۴) 12

۵ ـ با در نظر گرفتن گرامر مقابل کدام‌یک از گزینه‌های زیر صحیح است؟

$E \rightarrow T+E|T*E|T$
$T \rightarrow P-T|P$
$P \rightarrow F/P|F$
$F \rightarrow id|(E)$

۱) عملگرها دارای شرکت‌پذیری راست (Right associative) هستند و عملگر منها اولویت بیش‌تری نسبت به عملگر تقسیم دارد.
۲) عملگرها دارای شرکت‌پذیری راست (Right associative) هستند و عملگر منها اولویت کمتری نسبت به عملگر تقسیم دارد.
۳) عملگرهای ضرب و جمع اولویت بیشتری نسبت به سایر عملگرها دارند و عملگرها از سمت چپ شرکت‌پذیر (Left associative) هستند.
۴) عملگرهای ضرب و جمع تقدم کمتری نسبت به سایر عملگرها دارند و عملگرها از سمت چپ شرکت‌پذیر (Left associative) هستند.

۶ ـ گرامر دستورات شرطی مقابل رادر نظر بگیرید. با در نظر گرفتن ارتباط بخش else با نزدیک‌ترین then، کدام یک از گزینه‌های زیر صحیح است؟ (λ معرف رشته به طول صفر است).

$ST \rightarrow$ if BE Then ST EP
$ST \rightarrow$ other
$BE \rightarrow$ be
$EP \rightarrow$ else ST
$EP \rightarrow \lambda$

۱) با این گرامر به هیچ وجه نمی‌توان از تجزیه کننده LL(1) استفاده نمود.
۲) با این گرامر به هیچ وجه نمی‌توان از تجزیه‌کننده SLR (1) استفاده نمود.
۳) بدون نیاز به هیچ‌گونه تغییری در گرامر، می‌توان از هر دو تجزیه‌گر LL(1) و SLR (1) استفاده نمود.
۴) تجزیه زبان گرامر فوق بدون هیچ‌گونه تغییری در گرامر با استفاده از تجزیه‌گر LL(1) ممکن است، اما با استفده از تجزیه‌گر SLR (1) ممکن نیست.

آزمون کارشناسی ارشد سال ۱۳۸۳

۱ ـ گرامر G مفروض است (λ رشته‌ای به طول صفر است) کدام گزینه صحیح است؟

$$G: S \to Aa \mid Bb \mid A \to \lambda \mid B \to \lambda \mid A \to cAb \mid B \to dAa$$

۱) G یک گرامر $LL(1)$ است ولی $LALR(1)$ نیست.

۲) G یک گرامر $LL(1)$ نیست ولی $SLR(1)$ است.

۳) G یک گرامر $LL(1)$، $SLR(1)$ و $LALR(1)$ است.

۴) G یک گرامر $LL(1)$ و همچنین $LALR(1)$ است ولی $SLR(1)$ نیست.

۲ ـ در برنامه زیر هر دو کاربرد عدد 10.5 خطا است. خطای اول در Declaration و خطای دوم در عبارت به ترتیب در کدام‌یک از بخش‌های Compiler کشف می‌شوند؟

```
      ⋮
int   A[10.5]:
      ⋮
···   A[10.5]+B···
      ⋮
```

۱) Parser و Parser

۲) Parser و Scanner

۳) Parser و کد خطایاب در زمان اجرا

۴) Parser و یکی از Semantic routine ها

۳ ـ گرامر روبرو $SLR(1)$ نیست (ε رشته‌ای به طول صفر است) چون:

$$S \to Sb \mid aA$$
$$A \to bA \mid \varepsilon$$

۱) قانون تهی دارد.

۲) بازگشتی چپ است

۳) $b \in follow(A) \cap first(bA)$

۴) $follow(S) \cap follow(A) \neq \phi$

۴ ـ اگر برای گرامر روبرو جدول تجزیه $LL(1)$ را تشکیل دهیم در سطر A و ستون b خواهیم داشت:

$$S \to aAb \mid bB$$
$$A \to aA \mid \varepsilon$$

۱) pop

۲) $A \to \varepsilon$

۳) Error

۴) $a \to bB$

۵ ـ بسیاری از کامپایلرها برنامه ورودی را به کدهای 3 عملوندی ترجمه می‌کنند. در مواردی ممکن است ضرورتاً قسمتی از یک کد توسط یک روال مفهومی (Semantic routine) و بقیه آن کد توسط یک روال مفهومی دیگر تولید شود. در کدام یک از ساختارهای زیر هیچ کدی در دو قسمت و توسط دو روال مفهومی تولید نمی‌شود؟

۱) $\text{if } \langle \text{Boolean} - \text{expression} \rangle \text{ then } \langle \text{statement} \rangle$

۲) $\text{while } \langle \text{Boolean} - \text{expression} \rangle \text{ do } \langle \text{statement} \rangle$

۳) $\text{Repeat } \langle \text{statement} \rangle \text{ until } \langle \text{Boolean} - \text{expression} \rangle$

۴)
```
case ⟨expression⟩ do
⟨constant⟩ : ⟨statement⟩;
      ⋮
⟨constant⟩ : ⟨statement⟩;
end;
```

۶ ـ با توجه به این‌که در ساختارهای if – then – else تودرتو، هر else به نزدیک‌ترین if تعلق دارد، کدام یک از گزاره‌های زیر صحیح است؟

۱) برای ساختارهای مزبور گرامر غیرمبهم وجود ندارد.

۲) برای ساختارهای مزبور گرامر غیرمبهم وجود دارد ولی گرامر برای پارسر LL(1) مناسب نیست.

۳) برای ساختارهای مزبور گرامر غیرمبهم وجود دارد ولی گرامر برای پارسر SLR(1) مناسب نیست.

۴) برای تجزیه و تحلیل ساختارهای مزبور از هیچ یک از روش‌های پارسر LL(1)، SLR(1) به هیچ وجه نمی‌توان استفاده کرد.

آزمون کارشناسی ارشد سال ۱۳۸۴

۱ ـ قواعد گرامری روبرو را در نظر می‌گیریم. کدام‌یک از گزاره‌های زیر صحیح است؟

$D \rightarrow TL$

$T \rightarrow int / real$

$L \rightarrow L.id$

$L \rightarrow id$

۱) اسم و نوع متغیرها در حین تحلیلی معنایی وارد جدول نمادها می‌شوند.

۲) اسم و نوع متغیرهای معرفی شده به راحتی در مرحله تحلیل لغوی در جدول نمادها وارد می‌شوند.

۳) اسم متغیرها در حین تحلیل لغوی وارد جدول نمادها می‌شود ولی نوع آن‌ها در حین تولید کد است که تشخیص و وارد جدول می‌شود.

۴) با اجرا کردن قواعد معنایی (Sementic Rules) صحیح در حین یک تجربه پایین به بالا می‌توانیم نوع کلیه متغیرهای معرفی شده را مشخص و در جدول نمادها وارد کنیم.

۲ ـ می‌دانیم که هر زیر برنامه می‌تواند دارای تعدادی پارامتر باشد و هنگام فراخوانی زیر برنامه تعداد آرگومان‌ها باید به تعداد پارامترها مطابقت داشته باشد اگر در برنامه‌های این مطابقت رعایت نشده باشد خطای مربوطه در کدام یک از مراحل زیر ردیابی می‌شود؟

۱) زمان اجرا (Run Time)

۲) تحلیل نحوی (Syntax Analysis)

۳) تحلیل معنایی (Semantic Analysis)

۴) تولید کد (Code Generation)

۳ ـ کدام‌یک از گزاره‌های زیر در مورد زبان‌های برنامه‌نویسی C و پاسکال صحیح است؟

۱) هر دو از نوع حساس به متن هستند و به کمک گرامرهای حساس به متن تعریف شده‌اند.

۲) هر دو از نوع حساس به متن هستند ولی به وسیله گرامرهای مستقل از متن تعریف شده‌اند.

۳) هر دو از نوع مستقل از متن هستند و به همین دلیل آن‌ها را به وسیله گرامرهای مستقل از متن تعریف کرده‌اند.

۴) زبان پاسکال از نوع مستقل از متن و زبان C از نوع حساس به متن است و هر کدام توسط گرامری از نوع مربوطه تعریف شده‌اند.

۴ ـ دو declaration برای آرایه دوبعدی A، به‌ترتیب در زبان‌های C و پاسکال به شرح زیر مفروضند. برای هر متغیر از نوع T، 5 واحد حافظه لازم است.

T A[10][10]

A:array[1..11,-2..8]of T

برای محاسبهٔ آدرس A[I][J] و A[I,J]، دو کامپایلر C و پاسکال، بدون این‌که کدی برای بررسی داخل محدوده بودن اندیس‌ها تولید کنند:

۱) هر دو می‌توانند 4 کد میانی تولید کنند.

۲) هر دو حداقل 6 کد میانی تولید می‌کنند.

۳) به ترتیب حداقل 2 و 4 کد میانی تولید می‌کنند.

۴) به ترتیب حداقل 4 و 6 کد میانی تولید می‌کنند.

۵ ـ تعداد حالات نمودار SLR(1) و LALR(1) و حداکثر تعداد قواعد تولید در این حالات برای گرامر زیر به ترتیب چندتا است. فرض کنید حالت اولیه به صورت شکل زیر باشد و حالت پذیرش را نیز در تعداد کل حالات حساب کنید.(λ رشته‌ای به طول صفر است)

$D \rightarrow AB$
$A \rightarrow \lambda$ $\boxed{S \rightarrow \bullet\$D\$}$
$B \rightarrow b$

۱) 3,3,6,6 ۲) 3,3,7,7 ۳) 4,4,8,6 ۴) 4,4,7,7

۶ ـ قاعده تولید while به شکل زیر مفروض است. BE و بقیه ST ها با قواعد تولید دیگری توصیف می‌شوند.

$ST \rightarrow$ while BE do ST

می‌دانیم که در پارسرهای پایین به بالا، برای اختصاص روال‌های مفهومی (Semantic Routines) به قاعدهٔ تولید While، لازم است تعداد بیش‌تری قاعده تولید که مجموعاً معادل آن قاعده تولید باشند بنویسیم. تعداد آن‌ها حداقل چندتا است؟

۱) 5 ۲) 4 ۳) 3 ۴) 2

آزمون کارشناسی ارشد سال ۱۳۸۵

۱ ـ کد سه آدرسی (یعنی یک opcode، 2 آدرس عملوند و یک آدرس نتیجه) زیر توسط کامپایلر تولید شده است که در آن $(1 \le i \le 9)$ t_i حافظه‌های موقت‌اند. می‌خواهیم کد را فقط از جهت کاهش تعداد حافظه‌های موقت استفاده شده بهینه کنیم. حداقل تعداد حافظه‌های موقت در کد بهینه چند تا است؟

$+, a, b, t_1$
$-, c, d, t_2$
$*, t_1, t_2, t_3$
$+, t_3, k, t_4$
$+, e, f, t_5$
$-, g, h, t_6$
$*, t_4, t_5, t_7$
$+, t_6, J, t_8$
$/, t_4, t_8, t_9$

۱) 2
۲) 3
۳) 4
۴) 5

۲ ـ در کدام مورد لازم است از هر سه نوع آدرس‌دهی مستقیم بلافصل، و غیرمستقیم استفاده کرد؟ فرض کنید در زبان موردنظر برنامه فرعی یا تابع وجود ندارد؟

۱) تولید کد بهینه عبارات ریاضی که عملوندهای آن متغیر ساده و مقادیر ثابت باشند.
۲) تولید کد بهینه عبارات ریاضی که عملوندهای آن متغیر ساده و فیلدهای رکورد باشند.
۳) تولید کد بهینه عبارات ریاضی که عملوندهای آن فقط عناصر آرایه یک بعدی باشد و اندیس آرایه فقط متغیر ساده باشد.
۴) هیچ‌کدام

۳ ـ کدام‌یک از گزاره‌های زیر صحیح است؟

۱) هیچ گرامر مبهمی LL(1) نیست.
۲) هر گرامر غیرمبهمی LL(1) است.
۳) یک گرامر LL(1) است اگر و فقط اگر مبهم نباشد.
۴) هیچ ارتباطی بین مبهم بودن یک گرامر و LL(1) بودن آن وجود ندارد.

۴ ـ کدام‌یک از گرامرهای زیر LALR(1) است؟ گزینه کامل‌تر را انتخاب کنید.

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

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

۳) $E \to E+T \quad E \to E \ OR \ T \quad E \to T \quad T \to id \quad T \to (E)$

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

۱) الف ۲) الف و ب ۳) الف، ج و د ۴) الف، ب و د

۵ ـ گرامر زیر مفروض است و ε نشانهٔ رشته‌ای به طول صفر است. کدام گزینه در مورد این گرامر صحیح است؟

$S \to AB \mid bA$

$A \to aA \mid \varepsilon$

$B \to bB \mid \varepsilon$

۱) LL(1) نیست پس LALR(1) هم نیست.

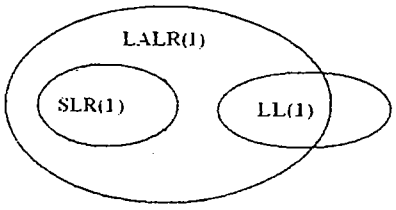
۲) LL(1) هست پس LALR(1) هم هست.

۳) LL(1) هست ولی LALR(1) بودن آن باید بررسی شود.

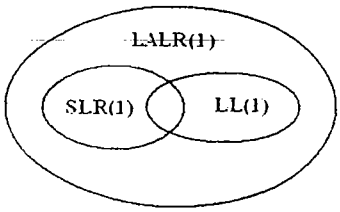
۴) LL(1) نیست ولی می‌توان معادلی از نوع LL(1) برای آن به‌دست آورد.

۶ ـ کدام دیاگرام مجموعه‌ای در مورد گرامرهای LL(1)، SLR(1) و LALR(1) صحیح است؟

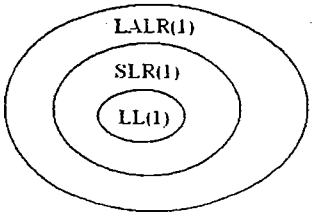
۱)



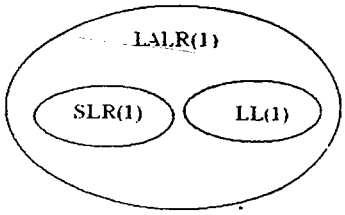
۲)



۳)



۴)



منابع

1) A. Aho, R. Sethi, and J. Ull man, Compilers: principles, reachniques, and tools, 1986
2) J. Tremblay and P. Sprenson, The Theory and Practice of Compiler Writing, 1985
3) C. Fisher a R. Le Blanc, Crafting a Compiler with C, 1991