به نام خدا

طراحی کامپایلر

آرش شفیعی

یک مترجم ساده

یک مترجم ساده

- برای معرفی مفاهیم و روش‌های اصول کامپایلر یک مترجم ساده را بررسی می‌کنیم.
- یک کامپایلر در دو مرحله یک برنامه را ترجمه می‌کند. در مرحلهٔ تحلیل [1] برنامه ورودی در زبان مبدأ به اجزایی تقسیم می‌شود و به یک کد میانی تبدیل می‌شود. در مرحلهٔ ترکیب [2] کد میانی به یک برنامه در زبان مقصد ترجمه می‌شود.

[1] analysis
[2] synthess

یک مترجم ساده

- ساختار نحوی [1] یک زبان برنامه‌نویسی فرم یا شکل کلی یک زبان برنامه‌نویسی را مشخص می‌کند و ساختار معنایی [2] یک زبان معنای برنامه‌های یک زبان را تعیین می‌کند.
- برای بیان ساختار نحوی یک زبان از یک نشانه‌گذاری به نام گرامر مستقل از متن یا فرم باکوس–نائور [3] استفاده می‌کنیم.
- یک روش برای ترجمه یک برنامه ترجمه نحوی [4] است که در این فصل معرفی خواهیم کرد.
- ترجمه نحوی روشی است برای پیاده‌سازی کامپایلر که توسط آن ترجمه زبان مبدأ توسط تجزیه‌کنندهٔ زبان (پارسر) [5] انجام می‌شود. در این روش به هر یک از قوانین گرامر یک معنی انتساب داده می‌شود.

---

[1] syntax
[2] semantics
[3] Backus–Naur Form
[4] syntax–directed translation
[5] parser

یک مترجم ساده

- با یک مترجم نحوی آغاز می‌کنیم که عبارات میانوندی را به عبارات پسوندی تبدیل می‌کند.
- برای مثال عبارت `2 + 5 - 9` به عبارت `+ 2 - 5 9` تبدیل می‌شود.
- قبل از شروع به ترجمه یک برنامه باید کلمه‌ها و نشانه‌های اصلی در برنامهٔ مبدأ مشخص شوند. برای این کار از تحلیل‌گر لغوی استفاده می‌کنیم.
- تحلیل‌گر لغوی کلمات یک برنامه که توکن نامیده می‌شوند را استخراج می‌کند. یک توکن دنباله‌ای از کاراکترها است که یک موجودیت واحد را مشخص می‌کند. برای مثال در عبارت `1 + count` شناسهٔ `count` عملگر `+` و عدد `1` هر کدام یک موجودیت واحد هستند. تحلیل‌گر لغوی همچنین فاصله‌های خالی [1] را تشخیص می‌دهد.

[1] white space

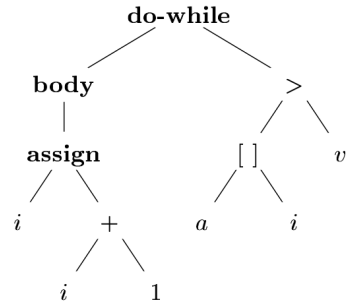
یک مترجم ساده

- یک برنامه، قبل از ترجمه به زبان مقصد به یک کد میانی تبدیل می‌شود.
- یکی از اشکال کد میانی، درخت تجزیه [1] است که ساختار نحوی یک برنامه را نمایش می‌دهد. پس از اینکه تحلیل‌گر لغوی توکن‌های یک برنامه را استخراج کرد، تحلیل‌گر نحوی یا تجزیه‌کننده یک درخت تجزیه از برنامه می‌سازد. این درخت تجزیه در نهایت به یک کد سه آدرسی تبدیل می‌شود.

[1] syntax tree

یک مترجم ساده

- در شکل زیر درخت تجزیه ساختار حلقه `do-while` در یک برنامه را مشخص می‌کند. این درخت تجزیه از عبارت `do i = i+1; while(a[i] > v);` به دست آمده است.
- ریشهٔ درخت، کل حلقه `do-while` ، فرزند سمت چپ بدنه حلقه و فرزند سمت راست شرط حلقه را تجزیه کرده است.



یک مترجم ساده

- یکی دیگر از اشکال کد میانی، دنباله‌ای است از دستورات سه آدرسی که در شکل زیر نمایش داده شده است.

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

یک مترجم ساده

- کدهای سه آدرسی برای محاسبه، یا مقایسه، و یا انشعاب به کار می‌روند.
- دستورات محاسباتی به صورت `x = y op z` نشان داده می‌شوند و در آن `op` یک عملگر محاسباتی و `x` و `y` و `z` سه عملوند هستند.
- دستورات مقایسه‌ای به صورت `if x relop y goto L` هستند که در آن `relop` یک عملگر مقایسه‌ای و `x` و `y` دو عملوند و `L` یک برچسب برای یکی از خطوط کد است.
- عملیات انشعاب نیز به صورت `goto L` به کار می‌روند جایی که `L` یک برچسب برای یکی از خطوط کد است.
- در کدهای سه آدرسی حداکثر سه عملوند وجود دارد و حداکثر یک عملیات انجام می‌شود که می‌تواند عملیات محاسباتی یا مقایسه‌ای باشد. در عملیات مقایسه‌ای تصمیم گرفته می‌شود که اجرای برنامه به کدام قسمت برنامه منتقل شود.

توصیف نحوی

- در این قسمت در مورد یک روش توصیفی به نام گرامرهای مستقل از متن [1] یا به اختصار گرامرها که برای توصیف نحوی یک زبان به کار می‌روند، صحبت خواهیم کرد.
- یک گرامر مجموعه قوانینی است که یک زبان را توصیف می‌کند. با استفاده از یک گرامر می‌توان تعیین کرد آیا یک برنامه به درستی در یک زبان برنامه نویسی توصیف شده است یا خیر. به عبارت دیگر با استفاده از گرامر یک زبان برنامه‌نویسی، می‌توان درستی یک برنامه نوشته شده در آن زبان را بررسی کرد.
- با استفاده از گرامر همچنین می‌توان اجزای یک برنامه را مشخص کرد.
- برای مثال دستور `if-else` در زبان جاوا به صورت زیر است.

```
if (expression) statement1 else statement2
```

[1] context-free grammar

توصیف نحوی

- بنابراین می‌توان گفت عبارت `if-else` از یک کلمه کلیدی `if` ، یک علامت پرانتز باز، یک عبارت، یک دستور (یا مجموعه‌ای از دستورات) ، کلمه کلیدی `else` و یک دستور (یا مجموعه‌ای از دستورات) تشکیل شده است.
- با استفاده از مجموعه‌ای از قوانین گرامری می‌توان تعیین کرد که آیا دستور فوق متعلق به زبان جاوا است یا خیر.
- با استفاده از قانون گرامر زیر برای زبان جاوا می‌توان دستور فوق را ساخت و بنابراین این دستور متعلق به زبان جاواست.

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

- یک قانون گرامر را قانون تولید [1] نیز می‌نامیم زیرا قسمتی از برنامه را تولید می‌کند.

[1] production rule

توصیف نحوی

- در یک قانون گرامر کلمات کلیدی مانند `if` و عملگرها مانند پرانتزها ترمینال‌ها [1] یا نمادهای پایانی نامیده می‌شوند و متغیرها مانند `stmt` و `expr` که می‌توانند با عبارات دیگر جایگزین شوند تا برنامه نهایی را بسازند، نمادهای غیرپایانی [2] یا متغیرهای گرامر نامیده می‌شوند.

[1] terminal
[2] nonterminal

تعریف گرامر

- یک گرامر مستقل از متن از چهار مؤلفه [1] تشکیل شده است.

۱. مجموعه‌ای از نمادهای پایانی یا ترمینال‌ها که توکن نیز نامیده می‌شوند. ترمینال‌ها کوچکترین اجزای تولید شده توسط گرامر هستند که به قسمت‌های کوچکتر تقسیم نمی‌شوند.

۲. مجموعه‌ای از نمادهای غیرپایانی یا متغیرهای نحوی [2] که قسمتی از برنامه را توسط دنباله‌ای از ترمینال‌ها و متغیرهای دیگر تعریف می‌کنند.

۳. مجموعه‌ای از قوانین تولید [3] به طوری که هر قانون از یک متغیر سمت چپ قانون، یک علامت پیکان یا فلش و دنباله‌ای از متغیرها و ترمینال‌ها (که بدنه یا سمت راست قانون نامیده می‌شود) تشکیل شده است.

۴. یک متغیر به نام متغیر یا نماد آغازین [4] .

---

[1] component
[2] syntatic variable
[3] production rule
[4] start symbol

تعریف گرامر

- یک گرامر را با مجموعه قوانین آن مشخص می‌کنیم. همه ارقام و اعداد و عملگرها و رشته‌هایی که معمولا به صورت پررنگ نوشته شده‌اند، ترمینال هستند. بقیه رشته‌ها نماد غیر پایانی یا متغیر هستند.
- سمت چپ برخی از قوانین یک گرامر می‌تواند مشابه باشد که در این صورت این قوانین در یک دسته قرار می‌گیرند و با علامت خط عمودی | از یکدیگر جدا می‌شوند.

تعریف گرامر

- گرامر زیر ساختار نحوی عبارات ریاضی که از عملگر جمع و تفریق تشکیل شده‌اند را توصیف می‌کند.

```
list → list + digit
list → list - digit
list → digit
digit → 0|1|2|3|4|5|6|7|8|9
```

تعریف گرامر

- بدنهٔ سه قانون تولید که از متغیر `list` تولید می‌شوند را می‌توان به طور خلاصه به صورت زیر نوشت.

```
list → list + digit | list - digit | digit
```

- ترمینال‌ها در این گرامر عبارتند از `0 1 2 3 4 5 6 7 8 9 - +`
- دو متغیر `list` و `digit` در این گرامر وجود دارند که متغیر `list` یک متغیر آغازین است.
- می‌گوییم یک قانون تولید متعلق به یک متغیر است اگر آن متغیر در سمت چپ آن قانون تولید باشد. رشته‌ای که از هیچ ترمینالی تشکیل نشده باشد را رشته تهی [1] می‌نامیم که با λ یا ϵ نمایش داده می‌شود.

[1] empty string

اشتقاق

- یک گرامر با شروع از متغیر آغازین و جایگزین کردن یک متغیر با بدنه قانون تولید متعلق به آن متغیر به طور مکرر یک رشته به دست می‌دهد یا مشتق [1] می‌کند. به فرایند به دست آوردن یک رشته (دنباله‌ای از ترمینال‌ها) از یک گرامر فرایند اشتقاق [2] می‌گوییم. همهٔ رشته‌هایی که از یک گرامر مشتق می‌شوند متعلق به زبان آن گرامر هستند.

[1] derives
[2] derivation

اشتقاق

- برای مثال فرایند اشتقاق برای به دست آوردن رشته 2 + 5 - 9 بدین صورت است.

```
list ⇒  list + digit
     ⇒  list - digit + digit
     ⇒  digit - digit + digit
     ⇒  9 - digit + digit
     ⇒  9 - 5 + digit
     ⇒  9 - 5 + 2
```

اشتقاق

- گرامر زیر فراخوانی یک تابع در زبان جاوا را توصیف می‌کند.

```
call → id(optparams)
optparams → params | ϵ
params → params, param | param
```

- در اینجا ϵ به این معناست که متغیر optparams می‌تواند در فرایند اشتقاق به رشته تهی تبدیل شود.

اشتقاق

- تجزیه [1] روشی است که توسط آن می‌توان یک رشته را از یک گرامر با شروع از متغیر آغازین مشتق کرد. در صورتی که نتوان یک رشته را از یک گرامر مشتق کرد تجزیه کننده خطای نحوی [2] صادر می‌کند.
- یکی از اساسی‌ترین مسائل در علم کامپایلر مسئلهٔ تجزیه است که با روش‌های مختلف آن آشنا خواهیم شد.

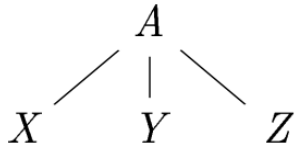
[1] parsing
[2] syntax error

درخت تجزیه

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

درخت تجزیه

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



درخت تجزیه

- یک درخت تجزیه [1] برای یک گرامر مستقل از متن درختی است که ویژگی‌های زیر را داراست.

۱- ریشه آن با متغیر آغازین بر چسب زده شده است.

۲- هر یک از برگ‌های آن با یک ترمینال یا رشته تهی برچسب زده شده‌اند.

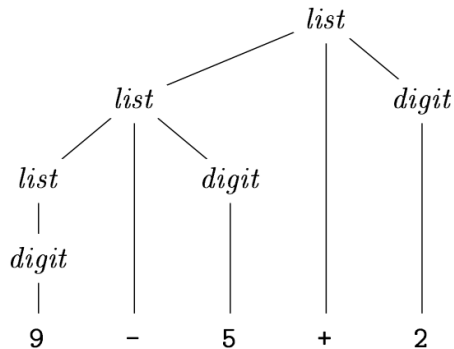
۳- هر یک از رئوس میان آن با یک متغیر برچسب زده شده است.

۴- اگر A یک متغیر باشد و X_1 و X_2 و $\cdots$ و X_n به ترتیب از سمت چپ فرزندان آن باشند، آنگاه باید قانون تولید $A \rightarrow X_1\ X_2\ \cdots\ X_n$ وجود داشته باشد. در اینجا هر یک از X_1 تا X_n ها می‌توانند یک ترمینال یا یک متغیر باشند. اگر قانون $A \rightarrow \epsilon$ وجود داشته باشد، آنگاه A تنها یک فرزند دارد که با ϵ برچسب زده شده است.

[1] parse tree

درخت تجزیه

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



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

درخت تجزیه

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

ابهام

- یک گرامر ممکن است برای یک رشته بیش از یک درخت تجزیه تولید کند. چنین گرامری یک گرامر مبهم [1] نامیده می‌شود.
- برای اینکه نشان دهیم یک گرامر مبهم است، کافی است رشته‌ای پیدا کنیم که برای آن بیش از یک درخت تجزیه وجود داشته باشد.
- به رشته‌ای که بیش از یک درخت تجزیه داشته باشد، می‌توان بیش از یک معنی منسوب کرد و بدین دلیل نمی‌توانیم در تهیه کامپایلر از یک گرامر مبهم استفاده کنیم. برای رفع ابهام باید گرامر را به گونه‌ای تغییر دهیم که برای رشته تنها یک درخت تجزیه وجود داشته باشد.

[1] ambiguous

ابهام

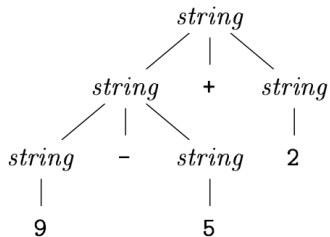
- فرض کنید گرامری به صورت زیر داریم :

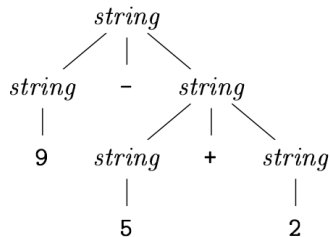
$$\texttt{string} \rightarrow \texttt{string + string} \mid \texttt{string - string} \mid \texttt{0}\mid\texttt{1}\mid\texttt{2}\mid\texttt{3}\mid\texttt{4}\mid\texttt{5}\mid\texttt{6}\mid\texttt{7}\mid\texttt{8}\mid\texttt{9}$$

- این گرامر همانند مثال قبل عبارات ریاضی با عملگرهای جمع و تفریق تولید می‌کند با این تفاوت که به جای متغیرهای `list` و `digit` از یک متغیر یکتای `string` استفاده شده است.

ابهام

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





- درخت سمت چپ در واقع 2 + (5 - 9) محاسبه می‌کند در حالی که درخت سمت راست (2 + 5) - 9 را محاسبه می‌کند. دو درخت تجزیه دو معنای متفاوت دارند.
- درخت تجزیه سمت راست مقدار 2 را برای این عبارت محاسبه می‌کند درحالی که مقدار عبارت برابر است با 6 .

وابستگی عملگرها

- به طور قراردادی `2 + 5 + 9` معادل است با `2 + (5 + 9)` و عبارت `2 - 5 - 9` معادل است با `2 - (9 - 5)`
- وقتی عملوند 5 دو عملگر + در سمت چپ و راست خود دارد، به طور قراردادی ابتدا عملگر سمت چپ را اعمال می‌کنیم. می‌گوییم عملگر + وابستهٔ چپ [1] است، زیرا عملوند 5 با دو عملگر + در سمت چپ و راست متعلق به عملگر سمت چپ است. در بیشتر زبان‌ها برنامه‌نویسی همهٔ عملگرهای حسابی جمع و تفریق و ضرب و تقسیم وابستهٔ چپ هستند.
- برخی از عملگرها مانند عملگر توان وابستهٔ راست هستند. برای مثال `2^3^4` معادل است با `2^(3^4)` .

[1] left-associative

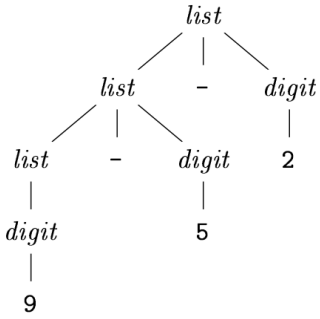
وابستگی عملگرها

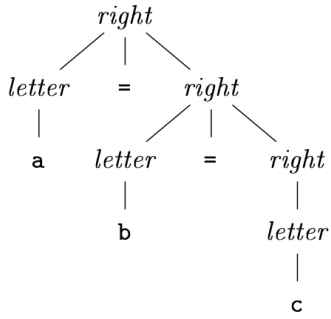
- به عنوان یک مثال دیگر، عملگر انتساب = در زبان سی وابستهٔ راست است، یعنی `a = b = c` معادل است با `a = (b = c)`.
- رشته‌هایی که از عملگر وابستهٔ راست انتساب تشکیل شده‌اند با استفاده از گرامر زیر به دست می‌آید.

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

وابستگی عملگرها

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





تقدم عملگرها

- عبارت 2 * 5 + 9 را در نظر بگیرید. این عبارت را می‌توانیم به دو صورت 2 * (5 + 9) و (2 * 5) + 9 تفسیر کنیم. قوانین وابستگی بر روی عملگرهای هم‌نوع (عملگرهایی که تقدم یکسانی دارند) اعمال می‌شوند، اما عملگر + و * دو عملگر متفاوت هستند.
- به طور قراردادی عملگر * اولویت یا تقدم بالاتری [1] نسبت به عملگر + دارد. بنابراین عبارت 2 * 5 + 9 به صورت (2 * 5) + 9 تفسیر می‌شود.

[1] higher precedence

تقدم عملگرها

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

```
+ - : left - associative
* / : left - associative
```

تقدم عملگرها

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

```
expr → expr + term | expr - term | term
```

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

تقدم عملگرها

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

```
term → term * factor | term / factor | factor
```

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

```
factor → digit | (expr)
```

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

تقدم عملگرها

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

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

تقدم عملگرها

- یک زیر مجموعه از دستورات زبان جاوا را می‌توانیم توسط گرامر زیر تجربه کنیم.

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

ترجمه نحوی

- ترجمه نحوی [1] روشی برای تحلیل معنایی یک برنامه است.
- در ترجمه نحوی به هر یک از قوانین گرامر یک معنی انتساب داده می‌شود. اگر معنی منتسب شده به قوانین گرامر در یک زبان مقصد باشند، درواقع در هنگام تحلیل نحوی زبان مبدأ به زبان مقصد ترجمه می‌شود و بدین دلیل به این روش ترجمه نحوی گفته می شود.
- برای مثال قانون تولید `expr` $\rightarrow$ `expr + term` را در نظر بگیرید. این قانون در یک گرامر مستقل از متن عباراتی تولید می‌کند که جمع چند عبارت هستند. اگر بخواهیم در مورد معنای این عبارت صحبت کنیم می‌گوییم مقدار عبارت `expr` در سمت چپ قانون برابر است با مقدار `expr` در سمت راست قانون به علاوه مقدار به دست آمده از عبارت `term`.

[1] syntax-directed translation

ترجمه نحوی

- برای اینکه بتوانیم `expr` در سمت چپ قانون را از `expr` در سمت راست متمایز کنیم، قانون را به صورت $\texttt{expr} \rightarrow \texttt{expr}_1 + \texttt{term}$ می‌نویسیم.
- اگر هر یک از متغیرهای این گرامر یک صفت یا ویژگی به نام مقدار (`value`) داشته باشند، می‌توانیم بنویسیم: $\texttt{expr.value} = \texttt{expr}_1\texttt{.value} + \texttt{term.value}$.
- در اینجا می‌خواهیم با استفاده از ترجمه نحوی عبارات ریاضی در فرم میانوندی [1] را به عبارات پسوندی [2] تبدیل کنیم.

[1] infix arthmetic expression
[2] postfix expression

ترجمه نحوی

- برای معرفی ترجمه نحوی باید دو مفهوم آشنا شویم :

۱- صفات یا ویژگی‌ها [1] : به هر یک از ساختارهای یک برنامه می‌توانیم یک ویژگی نسبت دهیم. برای مثال نوع یک عبارت در یک برنامه یا مقدار آن عبارت می‌توانند دو ویژگی از یک عبارت باشند. تعداد دستورات در یک بلوک برنامه می‌تواند یک ویژگی از یک بلوک برنامه باشد. از آنجایی که در گرامر مستقل از متن از نمادهای پایانی یا غیرپایانی (ترمینال‌ها و متغیرها) برای تجزیه و نمایش برنامه استفاده می‌کنیم به هر یک از نمادها در گرامر یک ویژگی یا صفت نسبت می‌دهیم.

۲- طرح کلی ترجمه [2] : طرح کلی ترجمه یک روش نشانه‌گذاری برای منتسب کردن معانی به قوانین گرامر است. ضمن تجزیه یک برنامه معانی منسوب شده به قوانین اجرا می‌شوند و تحلیل معنایی انجام می‌شود. می‌توان در این روند تحلیل معنایی، برنامه به زبانی دیگر ترجمه می‌شود.

[1] attributes
[2] translation scheme

ترجمه نحوی

- در این قسمت از ترجمه نحوی استفاده می‌کنیم تا عبارات محاسباتی میانوندی را به عبارت پسوندی تبدیل کنیم.
- یک عبارت پسوندی را می‌توانیم بدین صورت تعریف کنیم : (۱) اگر E یک متغیر یا ثابت باشد آنگاه عبارت پسوندی آن برابر با E است. (۲) اگر E عبارتی به صورت E_1 op E_2 باشد جایی که op یک عملگر است، آنگاه عبارت پسوندی آن E_1' E_2' op است، جایی که E_1' و E_2' معادل پسوندی E_1 و E_2 هستند. (۳) اگر E عبارتی پرانتز گذاری شده به صورت (E_1) باشد، آنگاه معادل پسوندی E برابر است با معادل پسوندی E_1.

ترجمه نحوی

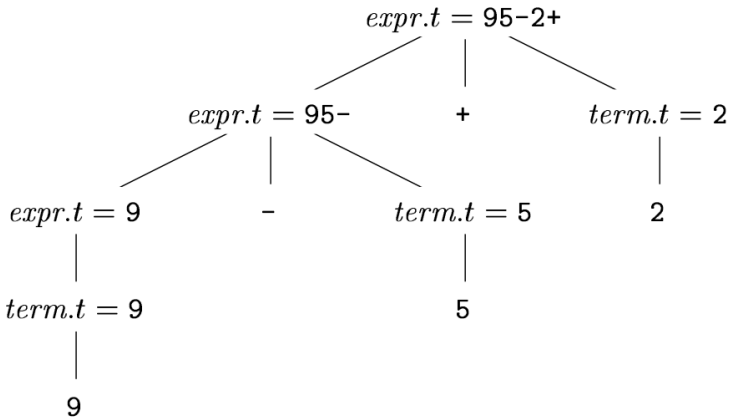
- برای مثال معادل پسوندی عبارت 2 + (5 − 9) برابر است با + 2 − 5 9 و معادل پسوندی (2 + 5) − 9 برابراست با − 4 2 5 9.
- فرض کنید می‌خواهیم عبارت * 3 − + 2 5 9 را محاسبه کنیم. از سمت چپ اولین عملگری که مشاهده می‌کنیم + است. این عملگر را در عبارت + 2 5 اعمال می‌کنیم و عبارت * + − 7 9 را به دست می‌آوریم. عملگر بعدی − است که پس از اعمال آن عبارت * 3 2 را به دست می‌آوریم. در نهایت مقدار محاسبه شده ضرب 2 و 3 است که مقدار 6 به دست می‌آید. بنابراین محاسبه عبارت‌های پسوندی بسیار ساده‌تر از محاسبه عبارات میانوندی است زیرا نیاز به اطلاعات در مورد تقدم وابستگی عملگرها ندارد.

ویژگی‌ها در ترجمه نحوی

- برای ترجمه نحوی به هریک از متغیرها و ترمینال‌های گرامر یک یا تعدادی ویژگی منتسب می‌کنیم. سپس به ازای هریک از قوانین نحوی یک یا تعدادی قوانین معنایی منتسب می‌کنیم تا با استفاده از آنها ویژگی‌ها را محاسبه کرده و برنامه را تحلیل کنیم.
- برای مثال فرض کنید می‌خواهیم عبارات میانوندی را به عبارات پسوندی تبدیل کنیم. در هنگام تجزیهٔ یک عبارت میانوندی ویژگی `t` معادل پسوندی آن عبارت است. پس معادل پسوندی عبارت `expr` برابراست با `expr.t`

ویژگی‌ها در ترجمه نحوی

- با تجزیهٔ عبارت 2 + 5 - 9 معادل پسوندی آن به صورت 2 5 - 9 + طبق درخت تجزیه زیر به دست می‌آید.



ویژگی‌ها در ترجمه نحوی

- یک ویژگی را ویژگی ساخته شده [1] می‌نامیم اگر مقدار آن در رأس N در درخت تجزیه از ویژگی فرزندان N در درخت به دست آید. برای محاسبه ویژگی‌های ساخته شده، درخت تجزیه از پایین به بالا پیمایش می‌شود. نوع دیگری از ویژگی‌ها ویژگی‌های به ارث برده شده نام دارند که مقدار آن در رأس N از ویژگی‌های پدر یا همزادان N به دست می‌آید. در مورد این ویژگی‌ها در فصل ترجمه نحوی بیشتر صحبت خواهیم کرد.

[1] synthesized

ویژگی‌ها در ترجمه نحوی

- حال می‌خواهیم گرامری بنویسیم که با استفاده از قوانین معنایی، یک عبارت میانوندی را به یک عبارت پسوندی تبدیل کند. ویژگی t معادل پسوندی یک عبارت را مشخص می‌کند.
- علامت || در ترجمه نحوی به معنای الحاق دو رشته به یکدیگر است.

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1$ + $term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1$ - $term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

ویژگی‌ها در ترجمه نحوی

- بنابراین اگر عبارتی به صورت $\texttt{expr} \rightarrow \texttt{expr}_1 + \texttt{term}$ داشته باشیم، معادل آن در نشانه‌گذاری پسوند برابراست با معادل پسوندی $\texttt{expr}_1$ الحاق شده به معادل پسوندی term الحاق شده به عملگر جمع.
بنابراین می‌نویسیم $\texttt{expr.t} = \texttt{expr}_1\texttt{.t} \parallel \texttt{term.t} \parallel '+'$
- در اینجا برای محاسبه ویژگی‌ها در درخت تجزیه از یک پیمایش عمق اول استفاده می‌کنیم، زیرا مقدار ویژگی‌ها از ویژگی‌های فرزندان به دست می‌آیند.

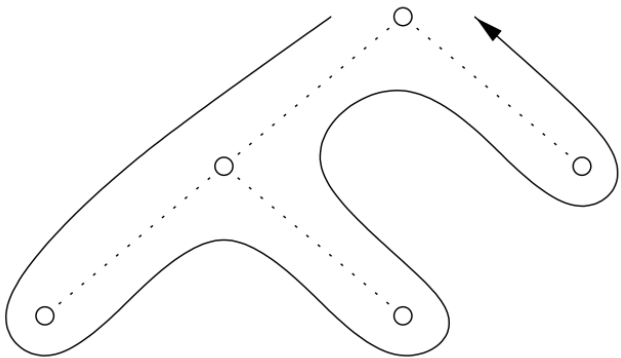
ویژگی‌ها در ترجمه نحوی

- در یک پیمایش عمق اول، همهٔ فرزندان یک رأس قبل از پیمایش آن رأس پیمایش می‌شوند.
- الگوریتم پیمایش عمق اول درخت تجزیه برای محاسبه قوانین معنایی به صورت زیر است.

```
procedure visit(node N) {
    for ( each child C of N, from left to right ) {
        visit(C);
    }
    evaluate semantic rules at node N;
}
```

ویژگی‌ها در ترجمه نحوی

- بنابراین برای محاسبه ویژگی‌ها با استفاده از پیمایش عمق اول یک درخت نمونه به صورت زیر پیمایش می‌شود.



تجزیه

- تجزیه [1] فرایندی است که تعیین می‌کند آیا یک رشته می‌تواند توسط یک گرامر تولید شود یا خیر. در فرایند تجزیه درخت تجزیه ساخته می‌شود.
- الگوریتم‌های زیادی برای تجزیه وجود دارند. یکی از این الگوریتم‌ها روش تجزیه کاهشی بازگشتی [2] است.
- برای هر گرامر مستقل از متن یک تجزیه کننده وجود دارد که در زمان $O(n^3)$ یک رشته با طول n را تجزیه کند. اما زمان مکعبی [3] معمولا بسیار پرهزینه است.
- برای دسته محدودتری از گرامرهای مستقل از متن تجزیه کننده‌هایی وجود دارند که در زمان خطی [4] یک رشته را تجزیه می‌کنند. برای تجزیه زبان‌های برنامه‌نویسی می‌توان از این تجزیه کننده‌های خطی استفاده کرد.

[1] parsing
[2] recursive descent parsing
[3] cubic time
[4] linear time

تجزیه

- دو دسته مهم از تجزیه کننده‌ها وجود دارند که تجزیه کننده‌های بالا به پایین [1] و پایین به بالا [2] نام دارند.
- در تجزیه کننده‌های بالا به پایین تجزیه از ریشه درخت تجزیه آغاز می‌شود و به سمت برگ‌ها حرکت می‌کند و در تجزیه پایین به بالا تجزیه از برگ‌ها آغاز می‌شود و به سمت ریشه حرکت می‌کند.
- تجزیه کننده‌های بالا به پایین معمولا برای پیاده‌سازی ساده‌تر هستند. تجزیه کننده‌های پایین به بالا پیچیده‌تر اند اما مجموعه بزرگ‌تری از گرامرها را می‌توانند تجزیه کنند و بنابراین بیشتر مورد استفاده قرار می‌گیرند.

[1] top-down
[2] bottom-up

تجزیه بالا به پایین

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

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

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

[1] top-down parser

تجزیه بالا به پایین

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

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

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

- در هرگام از فرایند تجزیه، تجزیه کننده با توجه به ترمینال (توکن) بعدی در رشته ورودی تصمیم می‌گیرد چه قانونی را انتخاب کند. به این ترمینال، ترمینال جلویی [1] گفته می‌شود.

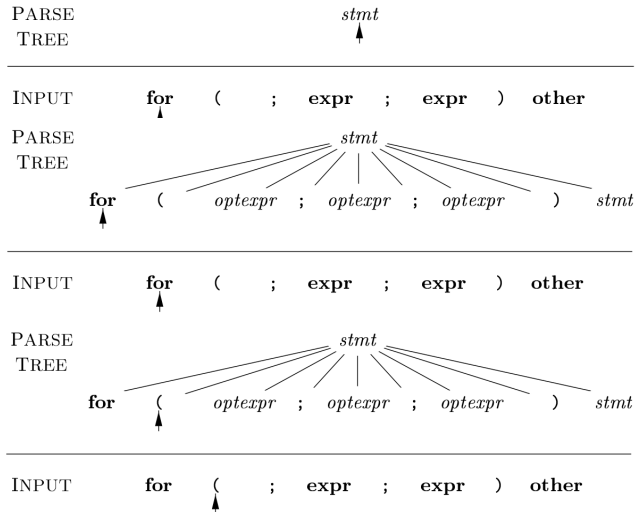
[1] lookahead

تجزیه بالا به پایین

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

تجزیه بالا به پایین

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



تجزیه بالا به پایین

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

[1] backtrack

تجزیه پیش‌بینی کننده

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

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

تجزیه پیش بینی کننده

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

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

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

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

تجزیه پیش بینی کننده

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

[1] match

تجزیه پیش بینی کننده

- تجزیه کنندهٔ پیش بینی کننده برای یک گرامر ساده می‌تواند مورد استفاده قرار بگیرد.
- برای تعریف گرامر ساده تابع `First(s)` را به صورت زیر تعریف می‌کنیم، جایی که `s` دنباله‌ای از ترمینال‌ها و متغیرهاست.
- اگر اولین کلمه در دنبالهٔ `s` ترمینال `t` باشد، آنگاه `First(s) = t`.
- اگر اولین کلمه در دنبالهٔ `s` متغیر `x` باشد و داشته باشیم $x \to s_1 \mid s_2 \mid \cdots \mid s_n$ آنگاه $\texttt{First(s)} = \texttt{First}(s_1) \cup \texttt{First}(s_2) \cup \cdots \cup \texttt{First}(s_n)$

تجزیه پیش بینی کننده

- حال فرض کنید در گرامر G داشته باشیم $A \to \alpha$ و $A \to \beta$. گرامر G ساده است اگر $\texttt{First}(\alpha) \cap \texttt{First}(\beta) = \emptyset$.
- از تجربه کننده پیش بینی کننده‌ای تنها زمانی می‌توان استفاده کرد که یک گرامر ساده باشد. در اینصورت در زمان خطی یک رشته را می‌توان تجزیه کرد.
- در تجزیه کننده پیش بینی کننده‌ای که طراحی کردیم، در پیاده‌سازی تابع `()optexpr` در صورتی که تطبیق رخ ندهد خطایی صادر نکردیم. با این کار در واقع قانون $\texttt{optexpr} \to \epsilon$ را پیاده‌سازی کردیم.

تجزیه پیش بینی کننده

- در حالت کلی اگر قانون به صورت $A \rightarrow X_1 \mid X_2 \mid \cdots \mid \epsilon$ داشته باشیم، پیاده‌سازی تابع `A()` هیچ خطایی صادر نمی‌کنیم، زیرا ممکن است رشته ورودی در بدنه هیچ یک از قوانین `A` منطبق نشود که در اینصورت قانون تهی اعمال می‌شود.

تجزیه پیش بینی کننده

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

بازگشت چپ

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

```
expr → expr + term
```

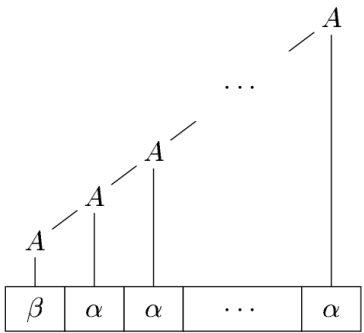
[1] recursive-descent parser
[2] left-recursive rule

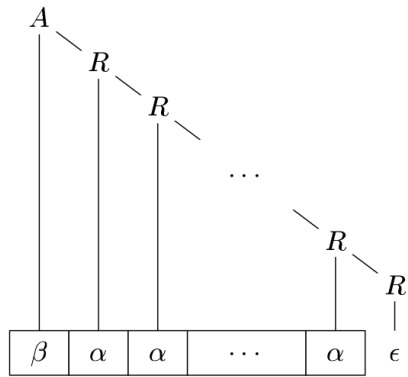
بازگشت چپ

- در این قانون متغیر سمت چپ قانون برابراست با اولین نماد در بدنهٔ قانون.
- حال در فرایند تجزیه اگر تابع `expr()` فراخوانی شود، این تابع نیز مجدداً تابع `expr()` را فراخوانی می‌کند و این فراخوانی بازگشتی خاتمه پیدا نمی‌کند.
- برای استفاده از تجزیه کننده کاهشی بازگشتی باید قوانین بازگشتی چپ را حذف کنیم.
- فرض کنید دو قانون تولید به صورت $A \rightarrow A\alpha|\beta$ داشته باشیم به طوری که α و β دنباله‌ای از ترمینال‌ها و متغیرها هستند که با A آغاز نمی‌شوند.
- دقت کنید ممکن است قانون $A \rightarrow A\alpha$ به طور مستقیم وجود نداشته باشد بلکه داشته باشیم $A \rightarrow X_1\alpha_1$ و $\cdots$ و $X_n \rightarrow A\alpha_n$ در این صورت نیز A یک قانون بازگشتی چپ محسوب می‌شود.

بازگشت چپ

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





بازگشت چپ

- گرامر جایگزین یک گرامر بازگشتی راست [1] است زیرا قانون $R \rightarrow \alpha R$ متغیر R را در سمت راست بدنهٔ قانون دارد. قوانین بازگشتی راست در تجزیهٔ کنندهٔ کاهشی بازگشتی مشکلی به وجود نمی‌آورند. در فصل‌های بعد روش کلی حذف بازگشت چپ را خواهیم دید.

[1] right recursive

تحلیل‌گر لغوی

- تحلیل‌گر لغوی [1] کاراکترها را از ورودی می‌خواند و آنها را گروه‌بندی کرده به صورت دنباله‌ای از توکن‌ها [2] درمی‌آورد.
- یک توکن علاوه بر کلمه‌ای که از ورودی خوانده است، نوع آن را نیز ذخیره می‌کند. بنابراین یک توکن درواقع یک ترمینال به علاوه اطلاعاتی اضافی است.
- دنباله‌ای از کاراکترها که یک توکن را تشکیل می‌دهند، یک کلمه [3] نامیده می‌شود.
- بنابراین یک تحلیل‌گر لغوی کلمات را از کاراکترها استخراج می‌کند و به عنوان ورودی به تجزیه کننده می‌دهد.

[1] lexical analyzer
[2] token
[3] lexeme

تحلیل‌گر لغوی

- تحلیل‌گر لغوی شناسه‌ها، اعداد، کلمات کلیدی، نمادها و عملگرها و فاصله‌های خالی را تشخیص می‌دهد و جداسازی می‌کند.
- در قانون تولید زیر num و id دو توکن هستند که نوع آنها به ترتیب عدد و شناسه است. می‌توانیم مقدار عدد یا رشته شناسه را مشاهده کنیم.

```
expr → expr + term | expr - term | term
term → term * factor | term / factor | factor
factor → (expr)
       | num {print(num.value)}
       | id {print(id.lexeme)}
```

تحلیل‌گر لغوی

- همچنین تحلیل‌گر لغوی از کاراکترهای فاصله خالی و خط جدید چشم‌پوشی می‌کند. البته نیاز است برای چاپ شماره خطی که در آن خطا رخ داده است، شماره خطوط نگهداری شود.

```
for ( ; ; peek = next input character ) {
        if ( peek is a blank or a tab ) do nothing;
        else if ( peek is a newline ) line = line+1;
        else break;
}
```

تحلیل‌گر لغوی

- تحلیل‌گر لغوی گاهی نیاز دارد کاراکترهای بعدی را بررسی کند تا یک کلمه را به درستی تشخیص دهد. برای مثال با خواندن کاراکتر < نمی‌توان مطمئن شد که علامت بزرگتر استخراج شده است، زیرا ممکن است کاراکتر بعدی = باشد و در نتیجه کلمه بعدی عملگر =< خواهد بود.
- معمولا تحلیل‌گر لغوی یک بافر داخلی دارد که در آن یک بلوک از رشته ورودی را نگهداری می‌کند، زیرا خواندن یک کاراکتر در هربار مراجعه به ورودی باعث کاهش سرعت می‌شود.

تحلیل‌گر لغوی

- برنامه زیر می‌تواند برای خواندن اعداد از رشته ورودی استفاده شود.

```
if ( peek holds a digit ) {
        v = 0;
        do {
                v = v * 10 + integer value of digit peek;
                peek = next input character;
        } while ( peek holds a digit );
        return token ⟨num, v⟩;
}
```

تحلیل‌گر لغوی

- یک تحلیل‌گر لغوی شناسه‌ها و کلمات کلیدی را نیز تشخیص می‌دهد و استخراج می‌کند. برای استخراج شناسه‌ها، تحلیل‌گر لغوی از توکن `id` استفاده می‌کند. بنابراین `count = count + inc;` را به صورت
  ⟨`id,"count"`⟩ ⟨`=`⟩ ⟨`id,"count"`⟩ ⟨`+`⟩ ⟨`id,"inc"`⟩ ⟨`;`⟩
- تحلیل‌گر لغوی لیستی از شناسه‌ها را در جدولی نگهداری می‌کند زیرا اجزای دیگر کامپایلر در فرایند کامپایل به شناسه‌ها و نوع آنها نیاز دارند. همچنین تحلیل‌گر لغوی برای شناسایی کلمات کلیدی از یک لیست تشکیل شده از کلمات کلیدی استفاده می‌کند.

تحلیل‌گر لغوی

- یک شناسه معمولا با یک کاراکتر آغاز می‌شود و با یک یا چند کاراکتر یا رقم ادامه می‌یابد. تحلیل‌گر لغوی به محض تشخیص یک شناسه در صورتی که شناسه در جدول شناسه‌ها موجود باشد، توکن آن شناسه را بازمی‌گرداند و در غیراینصورت یک توکن جدید ساخته به جدول شناسه‌ها اضافه می‌کند.

```
if ( peek holds a letter ) {
        collect letters or digits into a buffer b;
        s = string formed from the characters in b;
        w = token returned by words.get(s);
        if ( w is not null ) return w;
        else {
                Enter the key-value pair (s, ⟨id, s⟩) into words
                return token ⟨id, s⟩;
        }
}
```

تحلیل‌گر لغوی

- شمای کلی تحلیل‌گر لغوی یا اسکنر به صورت زیر است.

```
Token scan() {
        skip white space,
        handle numbers,
        handle reserved words and identifiers,
        /* if we get here, treat read-ahead character peek as a token */
        Token t = new Token(peek);
        peek = blank
        return t;
}
```

جداول علائم

- جداول علائم ساختارهای داده‌ای هستند که اطلاعاتی را در مورد ساختارهای برنامه نگهداری می‌کنند. این اطلاعات به مرور زمان در هر یک از بخش‌های کامپایلر تکمیل می‌شوند و در نهایت برای تولید کد مورد استفاده قرار می‌گیرند.

- محتوای این جداول معمولاً شناسه‌ها و متغیرها و نوع آنها و مکان آنها در حافظه را در برمی‌گیرند. برای مثال متغیرهای متمایز با نام مشابه را می‌توان توسط جداول علائم تمیز داد و یا نوع متغیرها را از جداول علائم استخراج کرد.

- معمولاً به ازای هریک از حوزه‌های تعریف در یک برنامه یک جدول علائم تشکیل داده می‌شود. یک حوزهٔ تعریف در زبان‌های رایج مانند سی و جاوا توسط علامت‌های آکولاد باز و بسته مشخص می‌شود. در زبان‌های شیءگرا برای هر کلاس و متغیرها و توابع آن یک جدول علائم تشکیل داده می‌شود.

- حوزهٔ تعریف [1] یک متغیر قسمتی از برنامه است که متغیر در آن قابل دسترسی است. در صورتی که بلوک‌های تودرتو در یک زبان مجاز باشند، چند متغیر با نام‌های یکسان می‌توانند در حوزه‌های تعریف تودرتو ساخته شوند.

جداول علائم

- فرض کنید زبانی داریم که حاوی بلوک‌های تودرتو و عباراتی جهت تعریف متغیرها و استفاده از متغیرهاست.
- برای مثال در این زبان می‌توانیم برنامه‌ای به صورت زیر داشته باشیم.

```
{int x; char y; {bool y; x; y; } x; y;}
```

- می‌خواهیم یک تحلیل‌گر معنایی داشته باشیم که با خواندن این برنامه رشته‌ای حاوی نوع متغیرهای استفاده شده در هر بلوک باشد. این تحلیل‌گر معنایی می‌تواند رشته زیر را تولید کند.

```
{{x : int; y : bool; } x : int; y: char; }
```

- کلاسی به نام Env تعریف می‌کنیم که درواقع یک جدول علائم است. یک جدول علائم حاوی اشاره‌گری که جدول علائم پدر است که در واقع اطلاعات بلوک پدر را نگهداری می‌کنیم. هنگام استخراج اطلاعات از جدول علائم، اگر متغیر مورد نظر در جدول موجود نباشد باید به ترتیب جداول علائم اجداد بررسی می‌شوند.

جداول علائم

- جدول علائم را به صورت زیر تعریف می‌کنیم :

```
1)  package symbols;
2)  import java.util.*;
3)  public class Env {
4)     private Hashtable table;
5)     protected Env prev;

6)     public Env(Env p) {
7)        table = new Hashtable(); prev = p;
8)     }

9)     public void put(String s, Symbol sym) {
10)       table.put(s, sym);
11)    }

12)    public Symbol get(String s) {
13)       for( Env e = this; e != null; e = e.prev ) {
14)          Symbol found = (Symbol)(e.table.get(s));
15)          if( found != null ) return found;
16)       }
17)       return null;
18)    }
19) }
```

جداول علائم

- شمای ترجمه برای زبان مذکور در زیر نشان داده شده است. در این شمای ترجمه از کلاس Env استفاده شده است.

$$
\begin{array}{rcll}
program & \rightarrow & & \{\ top = \mathbf{null};\ \} \\
 & & block & \\
 & & & \\
block & \rightarrow & '\{' & \{\ saved = top; \\
 & & & \quad top = \mathbf{new}\ Env(top); \\
 & & & \quad \text{print("\{ ");}\ \} \\
 & & decls\ stmts\ '\}' & \{\ top = saved; \\
 & & & \quad \text{print("\} ");}\ \} \\
 & & & \\
decls & \rightarrow & decls\ decl & \\
 & | & \epsilon & \\
 & & & \\
decl & \rightarrow & \mathbf{type}\ \mathbf{id}\ ; & \{\ s = \mathbf{new}\ Symbol; \\
 & & & \quad s.type = \mathbf{type}.lexeme \\
 & & & \quad top.put(\mathbf{id}.lexeme,\ s);\ \} \\
 & & & \\
stmts & \rightarrow & stmts\ stmt & \\
 & | & \epsilon & \\
 & & & \\
stmt & \rightarrow & block &
\end{array}
$$

جداول علائم

- اگر یک تجزیه کننده برای این گرامر نوشته شود و در هنگام تجزیه عملیات تحلیل معنایی در ترجمه نحوی انجام گیرد، به ازای ورودی :

```
{int x; char y; {bool y; x; y; } x; y;}
```

خروجی :

```
{{x : int; y : bool; } x : int; y: char; }
```

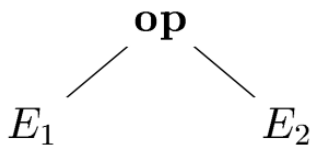
تولید می‌شود.

تولید کد میانی

- بخش تحلیل کامپایلر پس از تحلیل لغوی و نحوی و معنایی یک کد میانی تولید می‌کند. این کد میانی توسط بخش سنتر کامپایلر دریافت می‌شود و پس از بهینه‌سازی به کد زبان ماشین ترجمه می‌شود.
- دو کد میانی مهم عبارتند از درخت نحوی و کدهای سه آدرسی.

تولید کد میانی

- در یک درخت نحوی هریک از ساختارهای زبان یک رأس در درخت را تشکیل می‌دهند. برای مثال به ازای عبارت E_1 p E_2 درخت نحوی زیر را خواهیم داشت.



تولید کد میانی

- فرض کنید هر رأس درخت نحوی ساخته شده از نوع کلاس `Node` است. کلاس `Node` دو فرزند دارد : کلاس `Expr` که هر نوع عبارتی می‌تواند باشد و کلاس `Stmt` که درواقع یک دستور است. برای مثال کلاس `While` فرزند کلاس `Stmt` است.

تولید کد میانی

- شمای کلی ترجمه نحوی برای یک گرامر ساده به صورت زیر است.

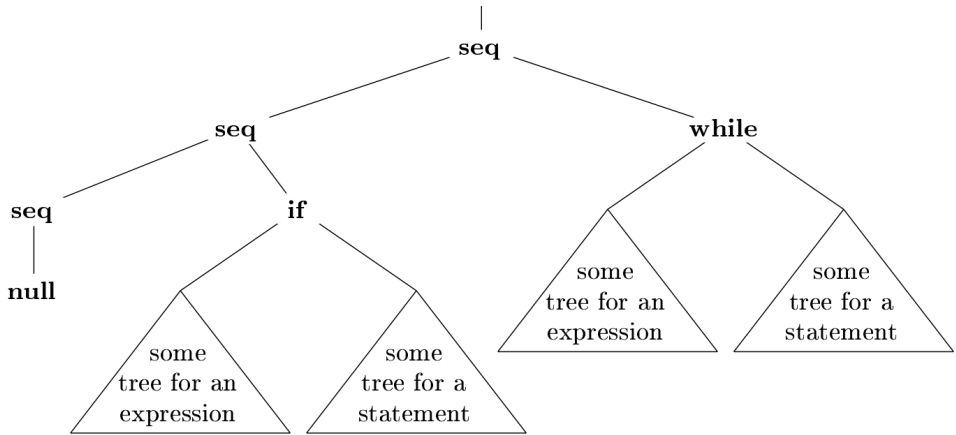
$$
\begin{array}{rcll}
\mathit{program} & \rightarrow & \mathit{block} & \{ \text{return } \mathit{block}.n; \} \\
\\
\mathit{block} & \rightarrow & \text{'\{'} \ \mathit{stmts} \ \text{'\}'} & \{ \mathit{block}.n = \mathit{stmts}.n; \} \\
\\
\mathit{stmts} & \rightarrow & \mathit{stmts}_1 \ \mathit{stmt} & \{ \mathit{stmts}.n = \textbf{new } \mathit{Seq}(\mathit{stmts}_1.n, \mathit{stmt}.n); \} \\
 & | & \epsilon & \{ \mathit{stmts}.n = \textbf{null}; \} \\
\\
\mathit{stmt} & \rightarrow & \mathit{expr} \ ; & \{ \mathit{stmt}.n = \textbf{new } \mathit{Eval}(\mathit{expr}.n); \} \\
 & | & \textbf{if} \ (\ \mathit{expr}\ ) \ \mathit{stmt}_1 & \\
 & & & \{ \mathit{stmt}.n = \textbf{new } \mathit{If}(\mathit{expr}.n, \mathit{stmt}_1.n); \} \\
 & | & \textbf{while} \ (\ \mathit{expr}\ ) \ \mathit{stmt}_1 & \\
 & & & \{ \mathit{stmt}.n = \textbf{new } \mathit{While}(\mathit{expr}.n, \mathit{stmt}_1.n); \} \\
 & | & \textbf{do} \ \mathit{stmt}_1 \ \textbf{while} \ (\ \mathit{expr}\ ) ; & \\
 & & & \{ \mathit{stmt}.n = \textbf{new } \mathit{Do}(\mathit{stmt}_1.n, \mathit{expr}.n); \} \\
 & | & \mathit{block} & \{ \mathit{stmt}.n = \mathit{block}.n; \} \\
\\
\mathit{expr} & \rightarrow & \mathit{rel} = \mathit{expr}_1 & \{ \mathit{expr}.n = \textbf{new } \mathit{Assign}(\text{'='}, \mathit{rel}.n, \mathit{expr}_1.n); \} \\
 & | & \mathit{rel} & \{ \mathit{expr}.n = \mathit{rel}.n; \} \\
\\
\mathit{rel} & \rightarrow & \mathit{rel}_1 < \mathit{add} & \{ \mathit{rel}.n = \textbf{new } \mathit{Rel}(\text{'}<\text{'}, \mathit{rel}_1.n, \mathit{add}.n); \} \\
 & | & \mathit{rel}_1 <= \mathit{add} & \{ \mathit{rel}.n = \textbf{new } \mathit{Rel}(\text{'}\leq\text{'}, \mathit{rel}_1.n, \mathit{add}.n); \} \\
 & | & \mathit{add} & \{ \mathit{rel}.n = \mathit{add}.n; \} \\
\\
\mathit{add} & \rightarrow & \mathit{add}_1 + \mathit{term} & \{ \mathit{add}.n = \textbf{new } \mathit{Op}(\text{'}+\text{'}, \mathit{add}_1.n, \mathit{term}.n); \} \\
 & | & \mathit{term} & \{ \mathit{add}.n = \mathit{term}.n; \} \\
\\
\mathit{term} & \rightarrow & \mathit{term}_1 * \mathit{factor} & \{ \mathit{term}.n = \textbf{new } \mathit{Op}(\text{'}*\text{'}, \mathit{term}_1.n, \mathit{factor}.n); \} \\
 & | & \mathit{factor} & \{ \mathit{term}.n = \mathit{factor}.n; \} \\
\\
\mathit{factor} & \rightarrow & (\ \mathit{expr}\ ) & \{ \mathit{factor}.n = \mathit{expr}.n; \} \\
 & | & \textbf{num} & \{ \mathit{factor}.n = \textbf{new } \mathit{Num}(\textbf{num}.\mathit{value}); \}
\end{array}
$$

تولید کد میانی

- در این مثال دستورات `while` ، `do` و `if` وجود دارند. در این مثال از `else` استفاده نکردیم، زیرا ایجاد ابهام می‌کند که در آینده درمورد آن توضیح خواهیم داد.
- همچنین دنباله‌ای از دستورات را با استفاده از کلاس `Seq` در درخت نحوی نشان می‌دهیم.

تولید کد میانی

- فرض کنید بلوکی داریم که اولین دستور آن یک دستور شرطی `if` و دومین دستور آن یک دستور حلقه تکرار `while` است. درخت نحوی متناظر آن به صورت زیر خواهد بود.



تولید کد میانی

- در ابتدای این فصل درمورد اولویت عملگرها صحبت کردیم و دیدیم که چگونه با افزودن متغیرها در یک گرامر، عملگرها با اولویت‌های متفاوت را تجزیه می‌کنیم.
- برای ساختن درخت تجزیه هنگامی که عملگرهای مشابه داریم می‌توانیم از کلاس‌های یکسان استفاده کنیم تا پیچیدگی کامپایلر و تعداد کلاس‌ها را کاهش دهیم. برای مثال عملگرهای + و * بسیار مشابه عمل می‌کنند و می‌توانیم از کلاس Op برای هر دو استفاده کنیم.

تولید کد میانی

- در جدول زیر عملگرهای رایج به همراه نوع انتزاعی آن نشان داده شده‌اند.

Concrete Syntax	Abstract Syntax
=	**assign**
||	**cond**
&&	**cond**
== !=	**rel**
< <= >= >	**rel**
+ -	**op**
* / %	**op**
!	**not**
$-_{unary}$	**minus**
[]	**access**

تولید کد میانی

- یک کامپایلر علاوه بر تحلیل نحوی، تحلیل معنایی نیز انجام می‌دهد. پس از تشکیل درخت نحوی کامپایلر در صورتی که خطاهای معنایی رخ دهد پیام خطا صادر می‌کند.
- برای مثال در صورتی که یک متغیر قبل از تعریف استفاده شود و یا دوباره تعریف شود و یا نوع متغیرها با عملیات انجام شده برروی آنها همخوانی نداشته باشد پیام خطا صادر می‌کند.