

به نام خدا

طراحی کامپایلر

آرش شفیعی



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 تحلیل¹ برنامه ورودی در زبان مبدأ به اجزایی تقسیم می‌شود و به یک کد میانی تبدیل می‌شود. در مرحلهٔ ترکیب² کد میانی به یک برنامه در زبان مقصد ترجمه می‌شود.

¹ analysis

² synthess

یک مترجم ساده

- ساختار نحوی¹ یک زبان برنامه‌نویسی فرم یا شکل کلی یک زبان برنامه‌نویسی را مشخص می‌کند و ساختار معنایی² 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-ناتور³ استفاده می‌کنیم.
- یک روش برای ترجمه یک برنامه ترجمه نحوی⁴ است که در این فصل معرفی خواهیم کرد.
- ترجمه نحوی روشی است برای پیاده‌سازی کامپایلر که توسط آن ترجمه زبان مبدأ توسط تجزیه‌کننده 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

- 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 برنامه 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 $count + 1$ شناسه `count` عملگر `+` و عدد `1` هر کدام یک موجودیت واحد هستند. تحلیل‌گر لغوی همچنین فاصله‌های خالی¹ را تشخیص می‌دهد.

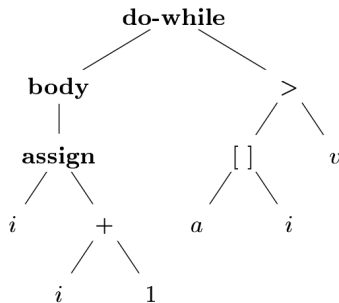
¹ white space

- یک برنامه، قبل از ترجمه به زبان مقصد به یک کد میانی تبدیل می‌شود.
- یکی از اشکال کد میانی، درخت تجزیه¹ 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.
- ریشه درخت، کل حلقه do-while، فرزند سمت چپ بدنه حلقه و فرزند سمت راست شرط حلقه را تجزیه کرده است.



- 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
```


- کدهای سه آدرسی برای محاسبه، یا مقایسه، و یا انشعاب به کار می‌روند.
- دستورات محاسباتی به صورت $x = y \text{ op } z$ نشان داده می‌شوند و در آن op یک عملگر محاسباتی و x و y و z سه عملوند هستند.
- دستورات مقایسه‌ای به صورت $\text{if } x \text{ relop } y \text{ goto } L$ 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 $\text{goto } L$ 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.

- در این قسمت در مورد یک روش توصیفی به نام گرامرهای مستقل از متن¹ 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 برنامه نویسی، 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

- 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` \rightarrow `if` (expr) `stmt` `else` `stmt`

- یک قانون گرامر را قانون تولید¹ نیز می‌نامیم زیرا قسمتی از برنامه را تولید می‌کند.

¹ production rule

- در یک قانون گرامر کلمات کلیدی مانند `if` و عملگرها مانند پرانتزها ترمینالها¹ یا نمادهای پایانی نامیده می‌شوند و متغیرها مانند `stmt` و `expr` که می‌توانند با عبارات دیگر جایگزین شوند تا برنامه نهایی را بسازند، نمادهای غیرپایانی² یا متغیرهای گرامر نامیده می‌شوند.

¹ terminal

² nonterminal

- 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 syntactic variable

³ production rule

⁴ start symbol

- یک گرامر را با مجموعه قوانین آن مشخص می‌کنیم. همه ارقام و اعداد و عملگرها و رشته‌هایی که معمولاً 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.

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

`list` \rightarrow `list + digit`

`list` \rightarrow `list - digit`

`list` \rightarrow `digit`

`digit` \rightarrow `0|1|2|3|4|5|6|7|8|9`

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

$$\text{list} \rightarrow \text{list} + \text{digit} \mid \text{list} - \text{digit} \mid \text{digit}$$

- 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 تهی¹ می‌نامیم که با λ یا ϵ نمایش داده می‌شود.

¹ empty string

- 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 مشتق¹ می‌کند. به فرایند به دست آوردن یک رشته (دنباله‌ای از ترمینال‌ها) از یک گرامر فرایند اشتقاق² می‌گوییم. همه رشته‌هایی که از یک گرامر مشتق می‌شوند متعلق به زبان آن گرامر هستند.

¹ derives

² derivation

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

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

- 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.

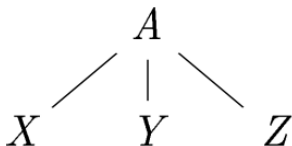
- تجزیه¹ روشی است که توسط آن می‌توان یک رشته را از یک گرامر با شروع از متغیر آغازین مشتق کرد. در صورتی که نتوان یک رشته را از یک گرامر مشتق کرد تجزیه کننده خطای نحوی² صادر می‌کند.
- یکی از اساسی‌ترین مسائل در علم کامپایلر مسئله تجزیه است که با روش‌های مختلف آن آشنا خواهیم شد.

¹ parsing

² syntax error

- 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 همه برنامه‌هایی که توسط گرامر آن زبان مشتق شوند. بنابراین رشته به دست آمده با شروع از متغیر آغازین یک گرامر، درواقع یک برنامه در آن زبان است.

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



- یک درخت تجزیه¹ برای یک گرامر مستقل از متن درختی است که ویژگی‌های زیر را داراست.

۱- ریشه آن با متغیر آغازین برچسب زده شده است.

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

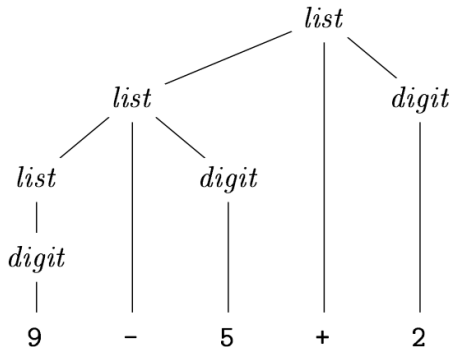
۳- هر یک از رئوس میانی آن با یک متغیر برچسب زده شده است.

۴- اگر A یک متغیر باشد و X_1 و X_2 و \dots و X_n به ترتیب از سمت چپ فرزندان آن باشند، آنگاه باید قانون تولید $A \rightarrow X_1 X_2 \dots X_n$ 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 $9 - 5 + 2$ از درخت تجزیه مشتق می‌شود.

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

- یک گرامر ممکن است برای یک رشته بیش از یک درخت تجزیه تولید کند. چنین گرامری یک گرامر مبهم¹ 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.

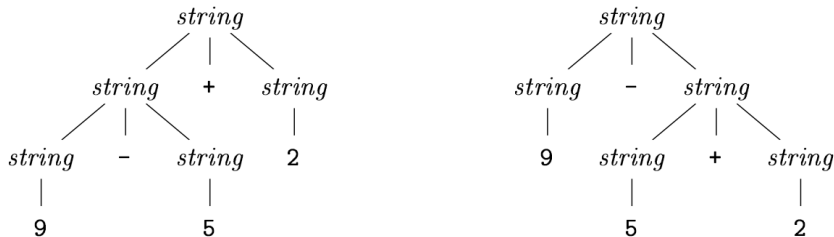
¹ ambiguous

- فرض کنید گرامری به صورت زیر داریم :

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

- 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.

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



- درخت سمت چپ در واقع $2 + (9 - 5)$ محاسبه می‌کند در حالی که درخت سمت راست $9 - (5 + 2)$ 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 $2 - (5 - 9)$
- وقتی عملوند 5 دو عملگر $+$ در سمت چپ و راست خود دارد، به طور قراردادی ابتدا عملگر سمت چپ را اعمال می‌کنیم. می‌گوییم عملگر $+$ وابسته¹ چپ است، زیرا عملوند 5 با دو عملگر $+$ در سمت چپ و راست متعلق به عملگر سمت چپ است. در بیشتر زبان‌ها برنامه‌نویسی همه عملگرهای حسابی جمع و تفریق و ضرب و تقسیم وابسته¹ چپ هستند.
- برخی از عملگرها مانند عملگر توان وابسته¹ راست هستند. برای مثال 2^{3^4} معادل است با $2^{(3^4)}$.

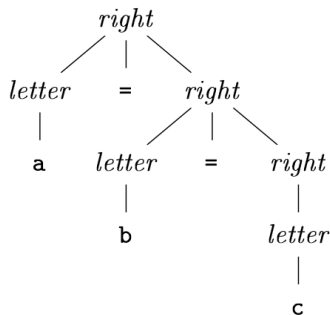
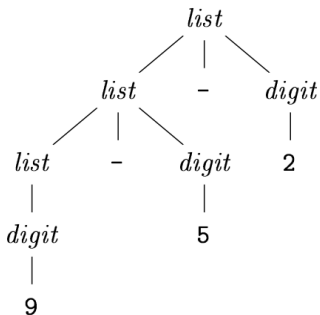
¹ left-associative

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

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

وابستگی عملگرها

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



- عبارت $9 + 5 * 2$ را در نظر بگیرید. این عبارت را می‌توانیم به دو صورت $2 * (9 + 5)$ و $(5 * 2) + 9$ 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 بالاتری¹ نسبت به عملگر $+$ دارد. بنابراین عبارت $9 + 5 * 2$ به صورت $(5 * 2) + 9$ تفسیر می‌شود.

¹ higher precedence

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

$+$ $-$: left-associative

$*$ $/$: left-associative

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.

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

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

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

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

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

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

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

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

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

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

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

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

- ترجمه نحوی¹ روشی برای تحلیل معنایی یک برنامه است. در ترجمه نحوی به هر یک از قوانین گرامر یک معنی انتساب داده می‌شود.
- برای مثال قانون تولید $\text{expr} \rightarrow \text{expr} + \text{term}$ 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

- برای اینکه بتوانیم expr در سمت چپ قانون را از expr در سمت راست متمایز کنیم، قانون را به صورت $\text{expr} \rightarrow \text{expr}_1 + \text{term}$ می‌نویسیم.
- اگر هر یک از متغیرهای این گرامر یک صفت یا ویژگی به نام مقدار (value) داشته باشند، می‌توانیم بنویسیم: $\text{expr.value} = \text{expr}_1.\text{value} + \text{term.value}$.
- در اینجا می‌خواهیم با استفاده از ترجمه نحوی عبارات ریاضی در فرم میانوندی¹ را به عبارات پسوندی² تبدیل کنیم.

¹ infix arithmetic expression

² postfix expression

- برای معرفی ترجمه نحوی باید با دو مفهوم آشنا شویم:

- ۱- صفات یا ویژگی‌ها¹ : 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 ترجمه² : 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.

¹ attributes

² translation scheme

- چگونه می‌توانیم تعداد دستورات در یک بلوک از برنامه را با استفاده از ترجمه نحوی محاسبه کنیم؟
- چگونه می‌توانیم نوع یک عبارت محاسباتی را در هنگام تجزیه با استفاده از ترجمه نحوی محاسبه کنیم؟

- 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 $E_1 \text{ op } E_2$ باشد جایی که op یک عملگر است، آنگاه عبارت پسوندی آن $E_1' E_2' \text{ op}$ 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 .

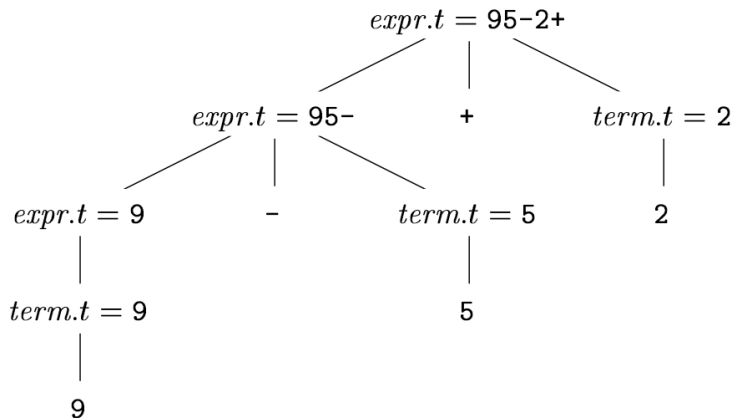
- برای مثال معادل پسوندی عبارت $(9 - 5) + 2$ برابر است با $9 - 5 + 2$ و معادل پسوندی $(5 + 2) - 9$ برابر است با $9 - 5 + 2$.
- فرض کنید می‌خواهیم عبارت $3 * 2 + 5$ را محاسبه کنیم. از سمت چپ اولین عملگری که مشاهده می‌کنیم $+$ است. این عملگر را در عبارت $2 + 5$ اعمال می‌کنیم و عبارت $3 * 7$ را به دست می‌آوریم. 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 تجزیه یک عبارت میانوندی ویژگی t معادل پسوندی آن عبارت است. پس معادل پسوندی عبارت $expr$ برابر است با $expr.t$

ویژگی‌ها در ترجمه نحوی

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



- یک ویژگی را ویژگی ساخته شده¹ 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 موروثی 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
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t term.t '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t term.t '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
\dots	\dots
$term \rightarrow 9$	$term.t = '9'$

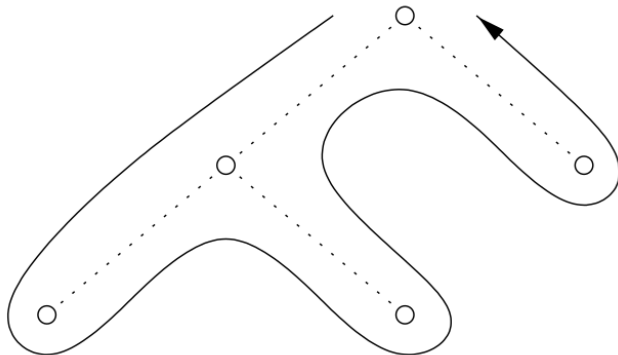
- بنابراین اگر عبارتی به صورت $\text{expr} \rightarrow \text{expr}_1 + \text{term}$ داشته باشیم، معادل آن در نشانه‌گذاری پسوندی برابر است با معادل پسوندی expr_1 الحاق شده به معادل پسوندی term الحاق شده به عملگر جمع.
بنابراین می‌نویسیم $\text{expr.t} = \text{expr}_1.\text{t} \parallel \text{term.t} \parallel '+'$
- در اینجا برای محاسبه ویژگی‌ها در درخت تجزیه از یک پیمایش عمق اول استفاده می‌کنیم، زیرا مقدار ویژگی‌ها از ویژگی‌های فرزندان به دست می‌آیند.

- در یک پیمایش عمق اول، همه 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.



- تجزیه¹ فرایندی است که تعیین می‌کند آیا یک رشته می‌تواند توسط یک گرامر تولید شود یا خیر. در فرایند تجزیه درخت تجزیه ساخته می‌شود.
- برای هر گرامر مستقل از متن الگوریتم تجزیه کننده سی‌وای کا² در زمان $O(n^3)$ یک رشته با طول n را تجزیه کند. اما زمان مکعبی³ معمولا بسیار پرهزینه است.
- برای دسته محدودتری از گرامرهای مستقل از متن تجزیه کننده‌هایی وجود دارند که در زمان خطی⁴ یک رشته را تجزیه می‌کنند. برای تجزیه زبان‌های برنامه‌نویسی می‌توان از این تجزیه کننده‌های خطی استفاده کرد.

¹ parsing

² Cocke–Younger–Kasami (CYK) algorithm

³ cubic time

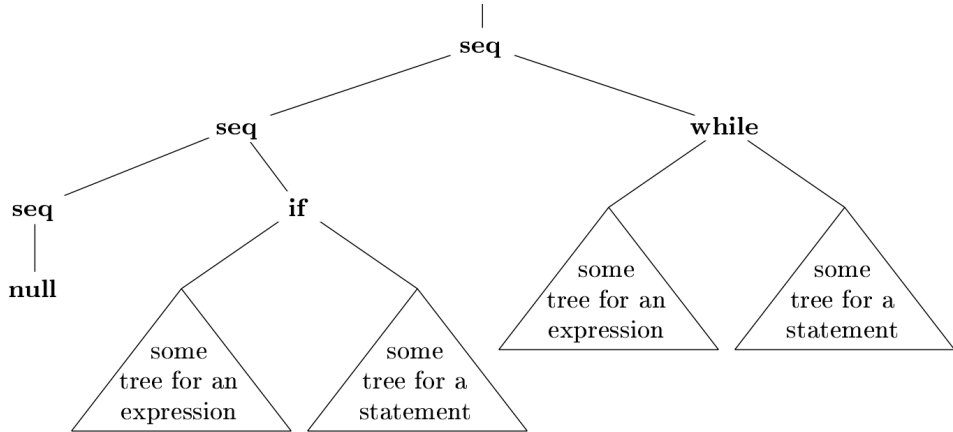
⁴ linear time

- دو دسته مهم از تجزیه کننده‌ها وجود دارند که تجزیه کننده‌های بالا به پایین¹ و پایین به بالا² 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 معمولاً 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

- 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.



تحلیل معنایی

- در هنگام تجزیه این درخت جدول علائم نیز تشکیل می شود و در هنگام تحلیل و پیمایش درخت، بررسی می شود که برنامه از لحاظ معنایی درست است.

