

به نام خدا

## زبان‌های برنامه‌نویسی

آرش شفیعی



## نحو و معناشناسی

- در این فصل با تعریف نحو و معناشناسی آغاز می‌کنیم. سپس روش‌های مهم برای توصیف نحوی را ارائه می‌کنیم. در مورد گرامرهای مستقل از متن، فرایند اشتقاق، درخت تجزیه، ابهام و تقدم عملگرها توضیح می‌دهیم.
- گرامرهای صفت<sup>1</sup> را که برای توصیف نحوی و معنایی در زبان‌های برنامه‌نویسی استفاده می‌شوند توضیح می‌دهیم. در انتها سه روش رسمی برای معناشناسی پویا، یعنی معناشناسی عملیاتی<sup>2</sup>، معناشناسی دلالتی<sup>3</sup>، و معناشناسی اصلی موضوعی<sup>4</sup> را توضیح می‌دهیم.

---

<sup>1</sup> attribute grammar

<sup>2</sup> operational semantics

<sup>3</sup> denotational semantics

<sup>4</sup> axiomatic semantics

- برای پیاده سازی درست یک زبان برنامه‌نویسی لازم است آن زبان را به طور دقیق و قابل فهم توصیف کنیم. برای زبان الگول در ابتدا توصیف دقیقی ارائه شد ولی آن توصیف قابل فهم نبود. توصیف درست زبان از اهمیت زیادی برخوردار است، زیرا کسانی که زبان را پیاده سازی می‌کنند باید آن را درست بفهمند و همچنین استفاده‌کنندگان زبان نیاز دارند ویژگی‌های زبان را به درستی بشناسند.
- مطالعه زبان‌های برنامه‌نویسی مانند مطالعه زبان‌های طبیعی می‌تواند به دو قسمت تقسیم شود: نحو‌شناسی<sup>1</sup> به مطالعه صورت عبارت در زبان مورد نظر و معناشناسی<sup>2</sup> به مطالعه معانی عبارات می‌پردازد.
- به عبارت دیگر در نحو‌شناسی مطالعه می‌کنیم چگونه کلمات ترکیب می‌شوند تا عبارات و جمله‌ها را بسازند، اما در معناشناسی به مطالعه معنی آن عبارات و جملات و درستی و نادرستی آنها می‌پردازیم.

---

<sup>1</sup> syntax

<sup>2</sup> semantics

- برای مثال در زبان جاوا ترکیب نحوی برای یک حلقه while به صورت زیر است.

`while (boolean-expr) statement`

- معنی این عبارت این است که تا وقتی که مقدار جمله `boolean-expr` درست است، دستورات عبارت `statement` را تکرار می‌شود و در صورتی که مقدار جمله `boolean-expr` نادرست ارزیابی شد، کنترل برنامه به بعد از حلقه منتقل می‌شود.

- یک زبان، چه طبیعی باشد مانند زبان انگلیسی و چه ساختگی مانند زبان جاوا، شامل رشته‌هایی است که از کاراکترهایی از یک الفبای معین تولید شده‌اند. به رشته‌های یک زبان، جمله نیز گفته می‌شود.
- قواعد نحوی یک زبان تعیین می‌کنند کدام رشته‌های تولید شده از یک الفبا متعلق به زبان هستند. زبان انگلیسی به طور مثال شامل یک مجموعه بزرگ و پیچیده از قواعد نحوی است که جملات زبان را تعیین می‌کنند. در مقایسه با زبان انگلیسی، حتی پیچیده‌ترین و بزرگ‌ترین زبان‌های برنامه‌نویسی قواعد نحوی ساده‌تر و کمتری دارند.

- توصیف رسمی قواعد نحوی زبان‌های برنامه‌نویسی، معمولاً نحوه تولید کوچک‌ترین اجزای زبان را مشخص نمی‌کند. کوچکترین اجزای یک جمله را تکواژ<sup>1</sup> یا لغت می‌نامیم. توصیف لغات را می‌توان به طور جداگانه با استفاده از یک توصیف‌کننده لغوی مشخص کرد. لغات در یک زبان برنامه‌نویسی شامل اعداد و ارقام، عملگرها، کلمات کلیدی و شناسه‌ها و اسامی می‌شود. در واقع می‌توانیم یک برنامه را مجموعه‌ای از لغات در نظر بگیریم.
- لغات را می‌توانیم به چند گروه تقسیم کنیم: برای مثال شناسه‌ها<sup>2</sup> شامل اسامی متغیرها، توابع، کلاس‌ها و غیره می‌شوند. برای هر گروه از لغات که در یک دسته قرار می‌گیرند، یک نام در نظر می‌گیریم که به آن توکن<sup>3</sup> می‌گوییم.

---

<sup>1</sup> lexeme

<sup>2</sup> identifier

<sup>3</sup> token

- عبارت  $\text{index} = 2 * \text{count} + 10$  در زبان جاوا را در نظر بگیرید.
- لغات و توکن‌های مربوط به این عبارت را می‌توانیم به صورت زیر نشان دهیم.

lexemes	tokens
index	identifier
=	equal-sign
2	int-literal
*	mult-op
count	identifier
+	plus-op
10	int-literal
;	semicolon



– زبان‌ها را می‌توان به دو روش توصیف کرد. به وسیله تشخیص<sup>1</sup> یا به وسیله تولید<sup>2</sup>.

---

<sup>1</sup> recognition

<sup>2</sup> generation

## تشخیص دهنده زبان

- فرض کنید زبان  $L$  را داریم که از الفبای  $\Sigma$  استفاده می‌کند. برای تعریف زبان  $L$  توسط تشخیص دهنده باید از یک مکانیزم  $R$  به عنوان دستگاه تشخیص دهنده زبان استفاده کنیم که قادر باشد رشته‌هایی که از الفبای  $\Sigma$  تشکیل شده است را دریافت و تشخیص دهد آیا آن رشته عضو زبان است یا خیر.  $R$  یا رشته را می‌پذیرد و یا رد می‌کند. این دستگاه تشخیص دهنده مانند فیلتری است که رشته‌های مجاز در آن زبان را از رشته‌های غیر مجاز جدا می‌کند. اگر  $R$  برای همه رشته‌ها بر روی الفبای  $\Sigma$ ، تنها جملات زبان  $L$  را پذیرفت آنگاه  $R$  یک توصیف برای زبان  $L$  است.
- تحلیل‌گر لغوی در یک کامپایلر در واقع یک تشخیص دهنده برای لغات زبانی است که کامپایل می‌کند. تحلیل‌گر لغوی، یک ورودی را که یک برنامه است دریافت کرده و لغات آن را تشخیص می‌دهد و استخراج می‌کند.

- یک تولید کننده زبان دستگاهی است که جملات یک زبان را تولید می کند.
- مکانیزمی که توسط آن یک زبان تولید می شود گرامر<sup>1</sup> نامیده می شود.
- توسط یک دستگاه تولید کننده زبان یا یک گرامر می توانیم بررسی کنیم آیا یک برنامه توسط آن گرامر قابل تولید است یا خیر.
- یکی از روش های توصیف قواعد نحوی، استفاده از گرامر است. معمولاً برای توصیف زبان از گرامر آن استفاده می کنیم.

---

<sup>1</sup> grammar

# گرامرهای مستقل از متن

- در اواسط دهه ۱۹۵۰ نوام چامسکی<sup>۱</sup> یکی از زبان شناسان مطرح، چهار دسته از گرامرها را برای تولید چهار دسته از زبانها توصیف کرد. دو دسته از این گرامرها به نام گرامرهای منظم<sup>۲</sup> و گرامرهای مستقل از متن<sup>۳</sup> برای توصیف نحوی زبانهای برنامه‌نویسی بسیار مورد استفاده قرار گرفتند.
- صورت توکن‌ها در زبانهای برنامه‌نویسی متعلق به دسته زبانهای منظم است و بنابراین لغات زبان را می‌توان توسط گرامر منظم توصیف کرده و توسط یک ماشین متناهی تشخیص داد. ساختار نحوی یک زبان برنامه‌نویسی متعلق به دسته زبانهای مستقل از متن است و می‌تواند توسط گرامرهای مستقل از متن توصیف شود.

---

<sup>۱</sup> Noam Chamsky

<sup>۲</sup> Regular

<sup>۳</sup> Context-free

- کمی بعد از انتشار تحقیقات چامسکی، جان باکوس<sup>1</sup> که عضو گروهی بود که بر روی زبان الگول کار می‌کردند، مقاله‌ای در مورد روش توصیف زبان‌های برنامه‌نویسی منتشر کرد. این روش جدید توصیف زبان که شبیه گرامر مستقل از متن بود، بعدها توسط پیتر ناور<sup>2</sup> کمی اصلاح شد. این فرم توصیف، بعدها به نام فرم باکوس-ناور<sup>3</sup> یا بی‌ان‌اف (BNF) مشهور شد.

---

<sup>1</sup> John Backus

<sup>2</sup> Peter Naur

<sup>3</sup> Backus-Naur form (BNF)

- یک فرا زبان <sup>1</sup> زبانی است که برای توصیف یک زبان دیگر استفاده می‌شود. یک گرامر در واقع یک فرا زبان برای زبان‌های برنامه‌نویسی است. گرامر در واقع ساختار نحوی یک زبان را مشخص می‌کند. برای مثال عبارت تخصیص مقدار یا انتساب <sup>2</sup> در زبان جاوا را می‌توانیم با مفهوم `<assign>` نمایش دهیم. می‌توانیم بنویسیم: `<assign> → <var> = <expression>`
- عبارت سمت چپ علامت فلش که `lhs` <sup>1</sup> نامیده می‌شود، مفهومی است که توسط این گرامر می‌خواهیم تعریف کنیم. عبارت سمت راست علامت فلش که `rhs` <sup>2</sup> نامیده می‌شود، تشکیل شده است از ترکیبی از توکن‌ها و مفاهیم دیگر. به طور کلی عبارت `lhs → rhs` را یک قانون تولید <sup>3</sup> می‌نامیم.

---

<sup>1</sup> metalanguage

<sup>2</sup> assignment expression

<sup>1</sup> left-hand side

<sup>2</sup> right-hand side

<sup>3</sup> production rule

- در مثالی که بیان شد مفاهیم `<var>` و `<expression>` باید تعریف شوند تا `<assign>` بتواند کاملاً تعریف شده و قابل استفاده باشد. با استفاده از قانونی که بیان شد می‌توانیم عبارت تخصیص مقدار زیر را بسازیم.

`total = t1 + t2`

- مفاهیم انتزاعی در گرامر را نمادهای غیر پایانی<sup>1</sup> یا متغیر و توکن را نمادهای پایانی<sup>2</sup> یا ترمینال می‌نامیم. یک گرامر از تعدادی قوانین تولید تشکیل شده است.

---

<sup>1</sup> nonterminal symbols

<sup>2</sup> terminal symbols

- یک نماد غیرپایانی یا متغیر معمولاً می‌تواند دو یا چند تعریف داشته باشد. چندین تعریف از یک نماد را می‌توانیم در چند قانون توصیف کنیم و یا اینکه در یک قانون توصیف و از علامت خط عمودی | در سمت راست قانون برای جداسازی قوانین متعدد استفاده کنیم.

- برای مثال عبارت if در جاوا را می‌توانیم به صورت زیر تعریف کنیم.

$\langle \text{if-stmt} \rangle \rightarrow \text{if } (\langle \text{logic-expr} \rangle) \langle \text{stmt} \rangle$

$\langle \text{if-stmt} \rangle \rightarrow \text{if } (\langle \text{logic-expr} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

- برای ترکیب دو قانون بالا می‌توانیم بنویسیم :

$\langle \text{if-stmt} \rangle \rightarrow \text{if } (\langle \text{logic-expr} \rangle) \langle \text{stmt} \rangle \mid \text{if } (\langle \text{logic-expr} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$



# گرامرهای مستقل از متن

- گرچه گرامرهای مستقل از متن ساده به نظر می‌رسند ولی با استفاده از آنها می‌توانیم همه قوانین نحوی زبان‌های برنامه‌نویسی را توصیف کنیم.
  - یک قانون می‌تواند بازگشتی باشد بدین معنی که مفهوم سمت چپ می‌تواند در سمت راست قانون نیز به کار رود. برای مثال برای تعریف یک لیست از شناسه‌ها می‌توانیم بنویسیم:
- `<id-list> → <id> | <id> ، <id-list>`
- در اینجا از یک قانون بازگشتی استفاده شده است تا بتوانیم یک لیست با هر طول دلخواهی را بسازیم.

- همانطور که گفتیم یک گرامر در واقع یک دستگاه تولیدکننده برای توصیف زبان است. یک جمله از یک زبان توسط دنباله‌ای از اعمال قوانین با شروع از یک نماد غیر پایانی که نماد آغازین<sup>1</sup> نامیده می‌شود، تولید می‌شود. این دنباله از اعمال قوانین را فرایند اشتقاق<sup>2</sup> می‌نامیم.
- معمولاً در یک زبان برنامه نویسی نماد آغازین <program> نامیده می‌شود که یک برنامه را توصیف می‌کند.

---

<sup>1</sup> start symbol

<sup>2</sup> derivation

- فرض کنید گرامر زیر یک زبان ساده را توصیف می‌کند.

$\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmt-list} \rangle \text{ end}$

$\langle \text{stmt-list} \rangle \rightarrow \langle \text{stmt} \rangle ; \mid \langle \text{stmt} \rangle ; \langle \text{stmt-list} \rangle$

$\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$

$\langle \text{var} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle \mid \langle \text{var} \rangle - \langle \text{var} \rangle \mid \langle \text{var} \rangle$

- یک فرایند اشتقاق به صورت زیر است.

$\langle \text{program} \rangle \Rightarrow \text{begin } \langle \text{stmt-list} \rangle \text{ end}$   
 $\Rightarrow \text{begin } \langle \text{stmt} \rangle ; \langle \text{stmt-list} \rangle \text{ end}$   
 $\Rightarrow \text{begin } \langle \text{var} \rangle = \langle \text{expression} \rangle ; \langle \text{stmt-list} \rangle \text{ end}$   
 $\dots$   
 $\Rightarrow \text{begin } A = B + C ; B = C ; \text{end}$

- علامت  $\Rightarrow$  را می‌خوانیم «به دست می‌دهد» یا «می‌دهد» یا «مشتق می‌کند».

- در هر مرحله از فرایند اشتقاق یکی از نمادهای غیر پایانی یا متغیرها با بدنه یکی قوانین متعلق به آن متغیر جایگزین می‌شود. هر یک از جملات به دست آمده در فرایند اشتقاق را یک صورت جمله‌ای<sup>1</sup> می‌نامیم.
- اگر در فرایند اشتقاق همیشه ابتدا اولین متغیر از سمت چپ جایگزین شود، آن فرایند را اشتقاق چپ<sup>2</sup> می‌نامیم. فرایند اشتقاق تا جایی ادامه پیدا می‌کند که هیچ متغیری در صورت جمله‌ای باقی نماند. یک صورت جمله‌ای که در آن هیچ متغیری نباشد را یک جمله می‌نامیم.
- فرایند اشتقاق می‌تواند از سمت راست نیز انجام شود، یعنی همیشه اولین متغیر سمت راست را جایگزین کنیم و یا فرایند اشتقاق ممکن است بدون هیچ ترتیبی انجام شود. ترتیب اشتقاق هیچ تأثیری بر روی زبان تولید شده توسط یک گرامر ندارد.

---

<sup>1</sup> sentential form

<sup>2</sup> leftmost derivation

- با یک جستجوی کامل بر روی گرامر می‌توان جملات یک زبان را یک به یک تولید کرد. البته اگر یک زبان نامحدود باشد تعداد جمله‌های آن نامحدود است و امکان تولید همه جملات وجود ندارد.
- گرامر زیر را در نظر بگیرید. این گرامر عبارات انتساب در یک زبان ساده را تعریف می‌کند.

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \langle \text{id} \rangle + \langle \text{expr} \rangle \mid \langle \text{id} \rangle * \langle \text{expr} \rangle \mid (\langle \text{expr} \rangle) \mid \langle \text{id} \rangle$

- برای مثال توسط این گرامر اگر می‌توانیم عبارت  $A = B * (A+C)$  را بسازیم.

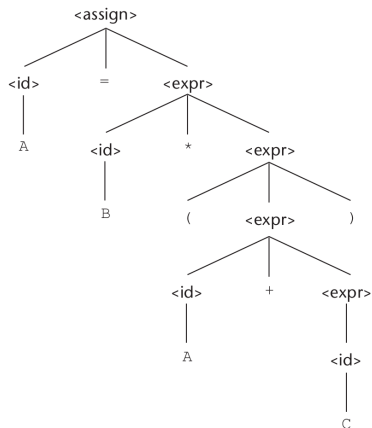
- گرامرها در واقع توسط یک ساختار سلسله مراتبی در فرایند اشتقاق جملات یک زبان را می‌سازند، بدین معنی که در هر سطح از سلسله مراتب یکی از متغیرها جایگزین می‌شود. این ساختار سلسله مراتبی درخت تجزیه<sup>1</sup> نامیده می‌شود.

---

<sup>1</sup> parse tree

## درخت تجزیه

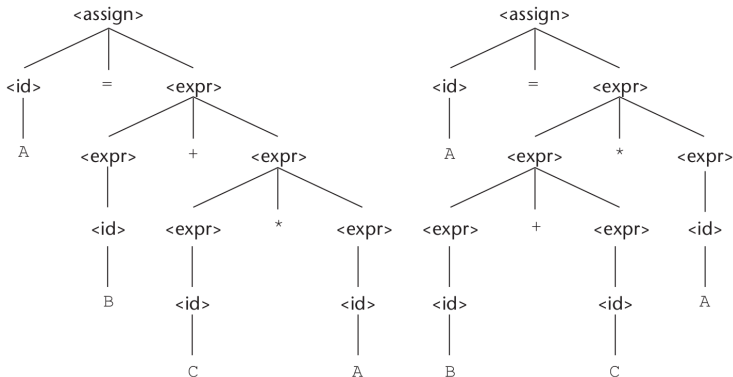
- برای مثال درخت تجزیه در شکل زیر نشان می‌دهد چگونه یک عبارت انتساب با استفاده از گرامر قبلی به دست می‌آید. هر یک از رئوس میانی در این درخت توسط متغیرها برچسب زده شده‌اند و هر برگ توسط یک ترمینال یا نماد پایانی برچسب زده شده‌است.





- تحلیل‌گر لغوی در یک کامپایلر، با دریافت برنامه ورودی، لغات آن را یک به یک استخراج می‌کند.
- در گام بعد، دنباله لغات به یک تحلیل‌گر نحوی داده می‌شود. توسط یک الگوریتم تجزیه که برای گرامر آن زبان طراحی شده است، در صورتی که برنامه از لحاظ نحوی درست باشد، یک درخت تجزیه تولید می‌شود و در غیر اینصورت پیام خطا صادر می‌شود.

- اگر یک جمله متعلق به یک گرامر، بتواند توسط بیش از یک درخت تجزیه تولید شود، به آن گرامر یک گرامر مبهم گفته می‌شود.
- گرامر قبلی برای تولید عبارات تخصیص مقدار را در نظر بگیرید. برای جمله  $A = B + C * A$  دو درخت تجزیه متفاوت وجود دارد.



- ابهام در یک گرامر ایجاد مشکل می‌کند، چرا که کامپایلرها معمولاً معانی جملات را از روی ساختار درخت به دست می‌آورند. در مثال قبل، کامپایلر برای محاسبه عبارت تخصیص مقدار، طبق درخت تجزیه، کد ماشین مورد نظر را تولید می‌کند. وقتی دو درخت تجزیه وجود داشته باشند، در واقع دو معنی برای یک عبارت وجود دارد و کامپایلر نمی‌تواند تصمیم بگیرد کدام معنی را انتخاب کند.
- به طور کلی اثبات شده‌است که هیچ الگوریتمی برای تعیین مبهم بودن یک گرامر وجود ندارد.
- اما برخی از ویژگی‌های یک گرامر می‌توانند تعیین‌کننده مبهم بودن گرامر باشند. اگر یک گرامر جمله‌ای را با دو اشتقاق چپ متفاوت یا دو اشتقاق راست متفاوت به دست بیاورد، آن گرامر مبهم است.
- در بسیاری از موارد یک گرامر را می‌توان به نحوی نوشت که مبهم نباشد. اگر نتوان یک گرامر را به نحوی نوشت که مبهم نباشد، زبان آن گرامر یک زبان ذاتاً مبهم است.

## تقدم عملگرها

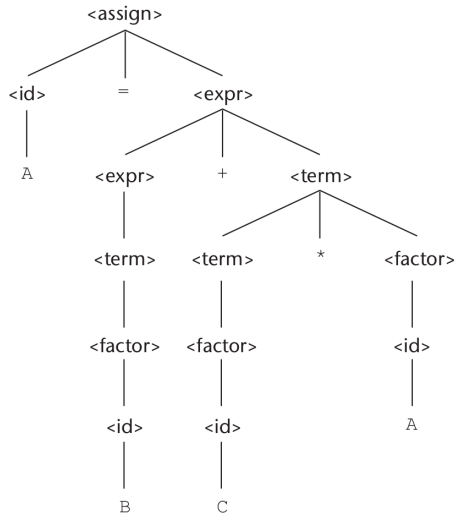
- وقتی در یک عبارت محاسباتی، چند عملگر وجود داشته باشد، یک مشکل معنایی که به وجود می‌آید، ترتیب ارزیابی عملگرهاست. برای مثال، در عبارت  $x + y * z$  آیا باید ابتدا عملگر جمع ارزیابی شود و یا عملگر ضرب؟
- بدین منظور، تقدم عملگرها تعریف می‌شوند. برای مثال، اگر تقدم ضرب بیشتر از جمع تعریف شود، آنگاه عملگر ضرب باید قبل از جمع ارزیابی شود.
- برای گرامر مبهم قبلی می‌توانیم یک گرامر غیر مبهم بنویسیم به طوری که درخت تجزیه ابتدا عملگر جمع و سپس عملگر ضرب را تجزیه کند. بدین ترتیب وقتی از برگ‌های درخت تجزیه برای ارزیابی یک عبارت آغاز می‌کنیم ابتدا عملگر ضرب را اعمال می‌کنیم.

- گرامر غیر مبهم زیر معادل گرامر مبهم قبلی است.

$$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$$
$$\langle \text{id} \rangle \rightarrow A \mid B \mid C$$
$$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{term} \rangle$$
$$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{factor} \rangle$$
$$\langle \text{factor} \rangle \rightarrow (\langle \text{term} \rangle) \mid \langle \text{id} \rangle$$

## تقدم عملگرها

- بدین ترتیب برای عبارت  $A = B + C * A$  تنها یک درخت تجزیه به صورت زیر وجود خواهد داشت.



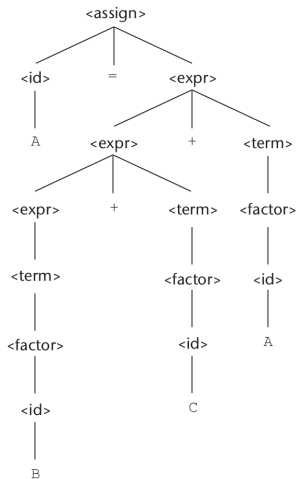
- وقتی در یک عبارت دو عملگر وجود داشته باشد که تقدم برابر داشته باشند، به قوانین معنایی نیاز داریم تا بدانیم کدام عملگر باید زودتر اجرا شود. وابستگی عملگرها<sup>1</sup> مشخص می‌کند که در شرایطی که تقدم یکسان است به کدام عملگر اولویت بالاتری داده می‌شود.

---

<sup>1</sup> operator associativity

## وابستگی عملگرها

- برای مثال عبارت  $A = B + C + A$  را در نظر بگیرید. درخت تجزیه برای این عبارت طبق گرامر غیر مبهمی که قبلاً ارائه شده به صورت زیر است.





## وابستگی عملگرها

- در این درخت تجزیه عملگر جمع اول زودتر محاسبه می‌شود. این عملیات صحیح است اگر وابستگی عملگر جمع از سمت چپ باشد که معمولاً هم همینطور است.
- در ریاضیات می‌گوییم عملگر جمع خاصیت شرکت پذیری<sup>1</sup> دارد، بدین معنی که وابستگی از چپ با وابستگی از راست معادل است، یعنی  $(A+B) + C = A + (B+C)$
- در کامپیوتر اما در عملیات جمع هم وابستگی می‌تواند مهم باشد، بدین معنی که محاسبات از چپ و راست می‌توانند نتایج متفاوتی تولید کنند. به طور مثال فرض کنید چند عدد را می‌خواهیم جمع کنیم و نتیجه را با دقت ۷ رقم اعشار محاسبه کنیم. اگر اولین عدد  $10^7$  باشد و بقیه اعداد ۱ باشند اضافه کردن اعداد ۱ تأثیری در نتیجه ندارد چرا که در رقم ۸ ام اضافه می‌شوند، و نتیجه در هر بار افزایش به میزان یک واحد برابر با  $10^7 \times 1.0000000$  خواهد بود.
- اما اگر ۱۰ عدد ۱ را با هم جمع کنیم و در نهایت با  $10^7$  جمع کنیم نتیجه  $10^7 \times 1.0000001$  خواهد شد.

---

<sup>1</sup> associative property

## وابستگی عملگرها

- در طراحی گرامر یک زبان برنامه نویسی وابستگی عملگرها باید در نظر گرفته شود.
- برای مثال در زبان سی تقدم عملگر \* و ++ یکسان است، اما وابستگی آنها از راست به چپ است، بدین معنا که اگر این دو عملگر در کنار یکدیگر قرار بگیرند، کامپایلر ابتدا عملگر سمت راست را محاسبه می‌کند.
- عبارت ++p ابتدا مقدار اشاره‌گر را افزایش می‌دهد و سپس مقدار آن را ارزیابی می‌کند، زیرا تقدم این دو عملگر یکسان و وابستگی آنها از راست به چپ است.
- اگر بخواهیم ابتدا مقدار اشاره‌گر را ارزیابی و سپس به آن یک واحد بیافزاییم، از عبارت ++(p) استفاده می‌کنیم.

- وقتی در یک گرامر، یک مفهوم یا متغیر lhs در یک قانون، در طرف چپ rhs باشد می‌گوییم این قانون بازگشتی چپ<sup>1</sup> است. یک قانون بازگشتی چپ تولید وابستگی از چپ می‌کند.
- به همین ترتیب اگر مفهوم lhs در یک قانون، در طرف راست rhs باشد می‌گوییم قانون بازگشتی راست<sup>2</sup> است. یک قانون بازگشتی راست تولید وابستگی از راست می‌کند.

$\langle \text{factor} \rangle \rightarrow \langle \text{expr} \rangle ** \langle \text{factor} \rangle \mid \langle \text{expr} \rangle$

$\langle \text{expr} \rangle \rightarrow (\langle \text{expr} \rangle) \mid \text{id}$

---

<sup>1</sup> left recursive

<sup>2</sup> right recursive

## گرامرهای غیر مبهم برای if-else

- قانون گرامر if را به صورت زیر در نظر بگیرید.

$\langle \text{if-stmt} \rangle \rightarrow \text{if}(\langle \text{logic-expr} \rangle) \langle \text{stmt} \rangle \mid \text{if}(\langle \text{logic-expr} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

- حال فرض کنید می‌خواهیم if های تو در تو داشته باشیم. اگر قانون  $\langle \text{if-stmt} \rangle \rightarrow \langle \text{stmt} \rangle$  را اضافه کنیم، این گرامر تبدیل به یک گرامر مبهم می‌شود.

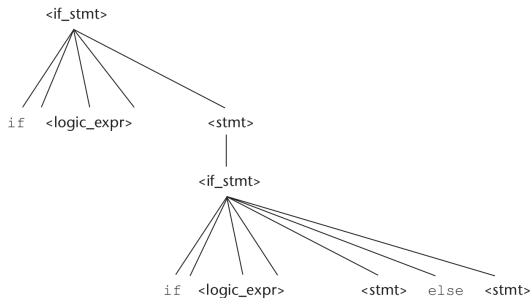
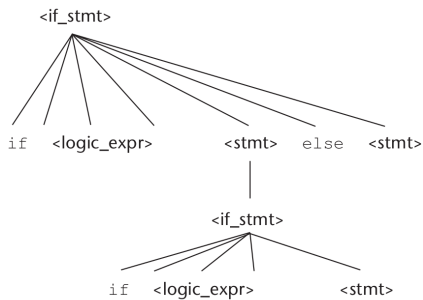
- یکی از صورت‌های جمله‌ای که توسط این گرامر ساخته می‌شود برابر است با

$\text{if}(\langle \text{logic-expr} \rangle) \text{ if}(\langle \text{logic-expr} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

- ابهام به وجود آمده این است که مشخص نیست else مربوط به کدام if است.

# گرامرهای غیر مبهم برای if-else

- دو درخت تجزیه برای این صورت جمله‌ای به شکل زیر هستند.



## گرامرهای غیر مبهم برای if-else

- کد زیر را در نظر بگیرید.

```
۱ if (done == true)
۲ if (denom == 0)
۳     quotient = 0 ;
۴     else quotient = num/denom ;
```

- اگر از درخت تجزیه اول (سمت چپ) استفاده شود، else وقتی اجرا می‌شود که مقدار done نادرست باشد.

## گرامرهای غیر مبهم برای if-else

- حال می‌خواهیم یک گرامر غیر مبهم برای if بنویسیم. قانون عبارات شرطی در همهٔ زبان‌های برنامه‌نویسی این است که else با نزدیک‌ترین if قبل از آن تطبیق داده می‌شود. بنابراین بین دو عبارت if و else نمی‌توان یک عبارت if بدون else گذاشت چرا که در غیر این‌صورت else با if دوم تطبیق داده می‌شود.
- برای حل این مشکل دو حالت در نظر می‌گیریم. حالتی که if بدون else باشد که باید در انتها یک بلوک if-else قرار بگیرد و حالتی که حالتی که if همراه با else باشد که در این‌صورت می‌توانیم if-else های تو در تو داشته باشیم.

`<stmt> → <matched> | <unmatched>`

`<matched> → if (<logic-expr>) <matched> else <matched> | <non-if-stmt>`

`<unmatched> → if (<logic-expr>) <stmt> | if (<logic-expr>) <matched> else`

`<unmatched>`

# گرامرهای مستقل از متن تعمیم یافته

- تعدادی روش جهت تعمیم گرامرهای مستقل از متن برای بهبود خوانایی آنها پیشنهاد داده شده‌اند.
  - در تعمیم اول، در سمت راست قانون می‌توانیم یک قسمت اختیاری قرار دهیم. هر عبارتی که در بین دو علامت براکت [ ] قرار بگیرد اختیاری است و می‌تواند وجود داشته باشد یا تهی باشد.
  - برای مثال در عبارت زیر قسمت آخر اختیاری است.
- `<if-stmt> → if (<expression>) <stmt> else [else <stmt>]`



## گرامرهای مستقل از متن تعمیم یافته

- در تعمیم دوم، در سمت راست عبارت می‌توانیم قسمتی را بین دو علامت آکولاد { } قرار دهیم، که بدین معنی است که عبارت بین آکولاد می‌تواند به هر تعداد بار دلخواه تکرار شود. بنابراین با استفاده از این روش می‌توانیم قانون‌های بازگشتی را ساده‌تر بنویسیم. برای مثال

$\langle \text{id-list} \rangle \rightarrow \langle \text{id} \rangle \{ , \langle \text{id} \rangle \}$

## گرامرهای مستقل از متن تعمیم یافته

- در تعمیم سوم، وقتی قسمتی از یک عبارت می‌تواند به چند حالت مختلف وجود داشته باشد. آن حالت‌ها را با استفاده از علامت | در بین دو علامت پرانتز ( ) از یکدیگر جدا می‌کنیم.

- برای مثال

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle ( * \mid / \mid \% ) \langle \text{factor} \rangle$

- گرامر مستقل از متن قانون بالا، به صورت زیر نوشته می‌شود.

$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{term} \rangle / \langle \text{factor} \rangle \mid \langle \text{term} \rangle \% \langle \text{factor} \rangle$

- بنابراین در گرامر مستقل از متن تعمیم یافته علامت‌های براکت، آکولاد و پرانتز جزء زبان گرامر هستند و ترمینال محسوب نمی‌شوند.

# گرامرهای مستقل از متن تعمیم یافته

- برای ساخت عبارت‌های ریاضی می‌توانیم از گرامر مستقل از متن تعمیم یافته زیر استفاده کنیم.

$$\langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle \{ (+ \mid -) \langle \text{term} \rangle \}$$
$$\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ (* \mid /) \langle \text{factor} \rangle \}$$
$$\langle \text{factor} \rangle \rightarrow \langle \text{exp} \rangle \{ ** \langle \text{exp} \rangle \}$$
$$\langle \text{exp} \rangle \rightarrow (\langle \text{expr} \rangle) \mid \langle \text{id} \rangle$$

# گرامرهای مستقل از متن تعمیم یافته

– قبلاً گفتیم که گرامرها یک دستگاه تولید کننده زبان هستند ولی می‌توانیم از آنها به عنوان تشخیص دهنده نیز استفاده کنیم. به ازای یک جمله باید الگوریتمی بنویسیم که بررسی کند آیا آن جمله توسط گرامر داده شده قابل تولید است یا خیر.

- یک گرامر صفت<sup>1</sup> وسیله ای است که با استفاده از آن ساختارهای دیگری که توسط گرامر مستقل از متن قابل توصیف نیستند، توصیف می‌شوند. در واقع یک گرامر صفت نوعی تعمیم برای گرامر مستقل از متن است.
- از گرامرهای صفت برای توصیف معانی جمله‌های تولید شده توسط گرامر استفاده می‌کنیم.

---

<sup>1</sup> attribute grammar

- برخی از ویژگی‌های زبان‌های برنامه‌نویسی نمی‌توانند توسط گرامرهای مستقل از متن بیان شوند. یکی از این ویژگی‌ها سازگاری نوع داده‌ها<sup>1</sup> است.
- برای مثال در زبان جاوا متغیرهای نوع اعدادی نمی‌توانند به متغیرهای نوع صحیح نسبت داده شوند، ولی برعکس آن امکان پذیر است. البته توصیف این ویژگی با استفاده از گرامر مستقل از متن امکان پذیر است، ولی باید تعدادی قوانین و نمادهای غیر پایانی به گرامر بیافزاییم که در این صورت گرامر زبان جاوا بسیار پیچیده و غیر قابل استفاده شود.

---

<sup>1</sup> type compatibility

- یک مثال دیگر از ویژگی‌های جاوا که ثابت شده است نمی‌توان آن را با گرامر مستقل از متن بیان کرد این است که همه متغیرها باید قبل از استفاده تعریف شده باشند.
- به این دلایل به قوانینی نیاز داریم که علاوه بر نحو، معنای عبارات را نیز توصیف کنند. به این دسته از قوانین، قوانین معناشناسی ایستا<sup>1</sup> می‌گوییم.
- در واقع قوانین معناشناسی ایستا مربوط به قوانین معنایی برنامه هستند که در زمان کامپایل قابل بررسی هستند.

---

<sup>1</sup> static semantic rules

- یکی از ابزارهایی که برای توصیف معناشناسی ایستا به کار می‌رود، گرامر صفت است که توسط دونالد کنوث برای توصیف نحو و معناشناسی ایستا ابداع شد.
- گرامرهای صفت تقریباً در همهٔ کامپایلرها به صورت غیر رسمی استفاده شده‌اند.
- معناشناسی پویا مربوط به معانی عبارات است که بعدها به آن اشاره خواهیم کرد.



- گرامرهای صفت در واقع گرامرهای مستقل از متن هستند که به آنها تعدادی صفت، توابع محاسبه صفت<sup>1</sup> و توابع مسندی<sup>2</sup> اضافه شده است.
- صفت‌ها به نمادهای گرامر ( نمادهای پایانی و غیر پایانی ) مربوط می‌شوند. توابع محاسبه صفت که توابع معنایی نیز نامیده می‌شوند، به قوانین گرامر مربوط می‌شوند. از این قوانین برای محاسبه صفت‌ها استفاده می‌شود. توابع مسندی، معنای قوانین را بیان می‌کنند و محدودیت‌هایی بر روی قوانین گرامر اعمال می‌کنند.

---

<sup>1</sup> attribute computation function

<sup>2</sup> predicate function

- به ازای هر نماد  $X$  در یک گرامر مجموعه‌ای از صفت‌ها به نام  $A(X)$  وجود دارد. مجموعه  $A(X)$  به دو مجموعه مجزا افزای می‌شود. مجموعه‌های  $S(X)$  و  $I(X)$  که صفت‌های ساخته شده<sup>1</sup> و صفت‌های ارث برده شده<sup>2</sup> نامیده می‌شوند.

---

<sup>1</sup> Synthesized attributes

<sup>2</sup> Inherited attributes

## تعریف گرامر صفت

- به ازای هر قانون گرامر، مجموعه‌ای از توابع معنایی وجود دارد.

- برای قانون

$$X_0 \rightarrow X_1 \cdots X_n$$

صفت‌های ساخته شده به صورت  $S(X_0) = f(A(X_1), \dots, A(X_n))$  محاسبه می‌شوند. بنابراین در یک درخت تجزیه صفت‌های ساخته شده با محاسبه صفت‌های فرزندان به دست می‌آید.

- صفت‌های به ارث برده شده به صورت  $I(X_j) = f(A(X_0), \dots, A(X_{j-1}), A(X_{j+1}), \dots, A(X_n))$  محاسبه می‌شوند. پس صفت‌های به ارث برده شده به صفت‌های پدر و همزادها<sup>3</sup> بستگی دارد.

- صفت‌های ساخته شده معانی را در درخت تجزیه به بالا منتقل می‌کنند در حالی که صفت‌های به ارث برده شده معانی را در درخت تجزیه به پایین منتقل می‌کنند.

---

<sup>3</sup> sibling

- یک تابع مسندی یک تابع منطقی است که مقدار آن درست یا نادرست است. این تابع محدودیت‌هایی بر روی قوانین گرامر اعمال می‌کند. نادرست بودن یک تابع مسندی، نشان دهندهٔ نقص در معنای عبارت است.
- یک درخت تجزیه برای یک گرامر صفت درخت تجزیه‌ای است که هر کدام از رئوس آن دارای مجموعه‌ای از صفت‌ها باشد که این مجموعه می‌تواند تهی نیز باشد.

- صفت های ذاتی<sup>1</sup> صفت های ساخته شده مربوط به برگ های درخت تجزیه هستند. برای مثال نوع یک متغیر در یک برنامه یک صفت ذاتی است که می توان آن را از جدول نمادها دریافت کرد. هنگامی که یک درخت تجزیه ساخته می شود، اولین ویژگی هایی که قابل محاسبه هستند، ویژگی های ذاتی برگ های درخت هستند. پس از آن، با استفاده از توابع صفت می توان صفت های رئوس دیگر را محاسبه نمود.

---

<sup>1</sup> intrinsic attributes

- برای مثال در زبان آدا<sup>1</sup>، نام یک تابع باید در پایان تعریف تابع نیز نوشته شود. این قید را نمی‌توان با استفاده از گرامر مستقل از متن بیان کرد. با استفاده از گرامر صفت این قانون را به صورت زیر بیان می‌کنیم.  
Syntax rule :  $\langle \text{proc-def} \rangle \rightarrow \text{procedure } \langle \text{proc-name} \rangle [1]$   
 $\langle \text{proc-body} \rangle \text{ end } \langle \text{proc-name} \rangle [2]$   
Predicate :  $\langle \text{proc-name} \rangle [1] \text{ string} = \langle \text{proc-name} \rangle [2] \text{ string}$
- دقت کنید هرگاه یک نماد غیر پایانی در سمت راست یک گرامر صفت تکرار شود، به ازای هر تکرار یک اندیس در بین دو براکت قرار می‌دهیم.
- در این گرامر صفت بیان کردیم که نام یک تابع که یک رشته بعد از کلمه کلیدی procedure است، باید با نام تابع که انتهای تعریف تابع، بعد از کلمه کلیدی end نوشته می‌شود، همخوانی داشته باشد.

---

<sup>1</sup> Ada

## مثال گرامر صفت

- حال یک مثال دیگر را در نظر می‌گیریم. در این مثال می‌خواهیم در یک زبان برنامه نویسی، عبارت‌های تخصیص مقدار داشته باشیم. نام متغیرها می‌تواند A یا B یا C باشد و نوع متغیرها می‌تواند int یا real باشد. سمت راست یک عبارت تخصیص مقدار می‌تواند یک متغیر و یا جمع چندین مقدار باشد. وقتی دو متغیر سمت راست از نوع‌های متفاوت باشند، مقدار محاسبه شده real است. اما وقتی دو متغیر سمت راست از یک نوع باشند، مقدار محاسبه شده از نوع آن متغیرهاست. نوع محاسبه شده در سمت راست عملیات انتساب باید با نوع متغیر سمت چپ عملیات انتساب یکسان باشد.
- با استفاده از گرامر مستقل از متن، این گرامر را به صورت زیر می‌نویسیم :

$\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle \mid \langle \text{var} \rangle$

$\langle \text{var} \rangle \rightarrow A \mid B \mid C$

- حال برای متغیرهای این گرامر دو متغیر صفت در نظر می‌گیریم. نوع واقعی (actual-type) و نوع مورد انتظار (expected-type).
- نوع واقعی : هرکدام از متغیرهای `<var>` و `<expr>` در گرامر، یک صفت ساخته شده دارند که برای ذخیره نوع آنها (که `int` یا `real` است) به کار می‌رود. برای نماد غیر پایانی `<var>` صفت آن ذاتی است و برای نماد غیر پایانی `<expr>` صفت آن (که نوع داده‌ای آن است) از روی صفت فرزندان آن به دست می‌آید.
- نوع مورد انتظار : نوع مورد انتظار، یک صفت به ارث برده شده برای نماد غیر پایانی `<expr>` است. در واقع انتظار می‌رود نوع `<expr>` در یک عبارت انتساب، با نوع متغیر `<var>` یکسان باشد.



## مثال گرامر صفت

– گرامر صفت برای مثال قبلی را به صورت زیر می‌نویسیم.

- Syntax rule :     $\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

Semantic rule :     $\langle \text{expr} \rangle . \text{expected-type} \leftarrow \langle \text{var} \rangle . \text{actual-type}$
- Syntax rule :     $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle [2]^+ \langle \text{var} \rangle [3]$

Semantic rule :     $\langle \text{expr} \rangle . \text{actual-type} \leftarrow \text{if } ( \langle \text{var} \rangle [2] . \text{actual-type} = \text{int} )$   
                                and (  $\langle \text{var} \rangle [3] . \text{actual-type} = \text{int}$  )  
                                then int  
                                else real  
                                and if

Predictor :     $\langle \text{expr} \rangle . \text{actual-type} = \langle \text{expr} \rangle . \text{expected-type}$
- Syntax rule :     $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle$

Semantic rule :     $\langle \text{expr} \rangle . \text{actual-type} \leftarrow \langle \text{var} \rangle . \text{actual-type}$

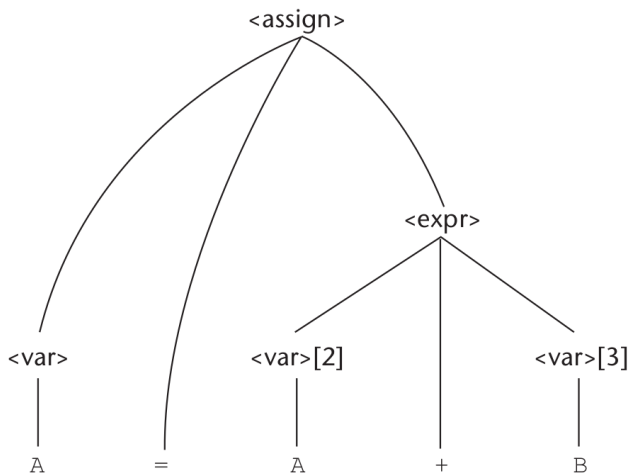
Predictor :     $\langle \text{expr} \rangle . \text{actual-type} = \langle \text{expr} \rangle . \text{expected-type}$
- Syntax rule :     $\langle \text{var} \rangle \rightarrow A \mid B \mid C$

Semantic rule :     $\langle \text{var} \rangle . \text{actual-type} \leftarrow \text{look-up } ( \langle \text{var} \rangle . \text{string} )$

- تابع look-up در واقع به ازای نام یک متغیر، نوع آن را باز می‌گرداند.

## مثال گرامر صفت

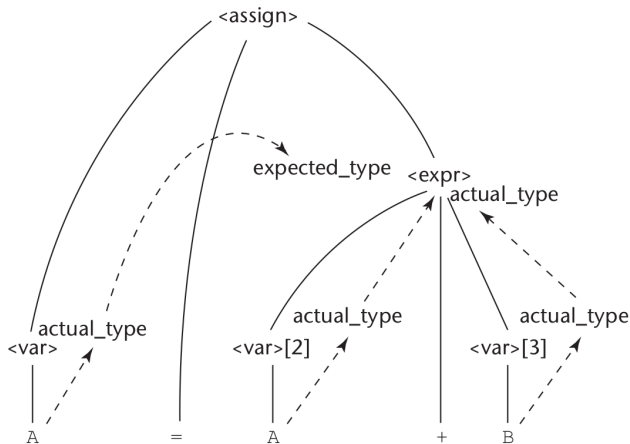
- درخت تجزیه برای عبارت  $A = A + B$  در شکل زیر نشان داده شده است.



- حال فرایند محاسبه صفت‌ها در یک درخت تجزیه را در نظر بگیرید. اگر همهٔ صفت‌ها، صفت‌های به ارث برده شده بودند، با پیمایش درخت از بالا به پایین می‌توانیم صفت همه رئوس را بدست آوریم. اگر همه صفت‌ها، صفت‌های ساخته شده بودند با پیمایش درخت از پایین به بالا می‌توانیم همه صفت‌ها را محاسبه کنیم. اما در واقع همیشه ترکیبی از صفت‌های به ارث برده شده و ساخته شده داریم پس پیمایش از هر دو طرف صورت می‌گیرد.

## مثال گرامر صفت

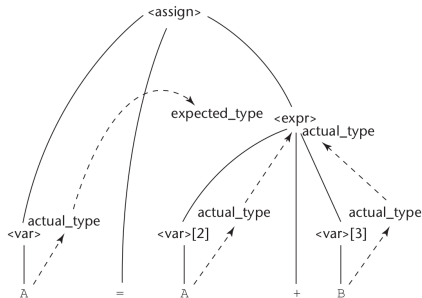
- شکل زیر نحوه محاسبه صفت‌ها در درخت تجزیه را نشان می‌دهد. نوع واقعی (actual-type) یک صفت ساخته شده است، درحالی که نوع مورد انتظار (expected-type) یک صفت به ارث برده شده است.



## مثال گرامر صفت

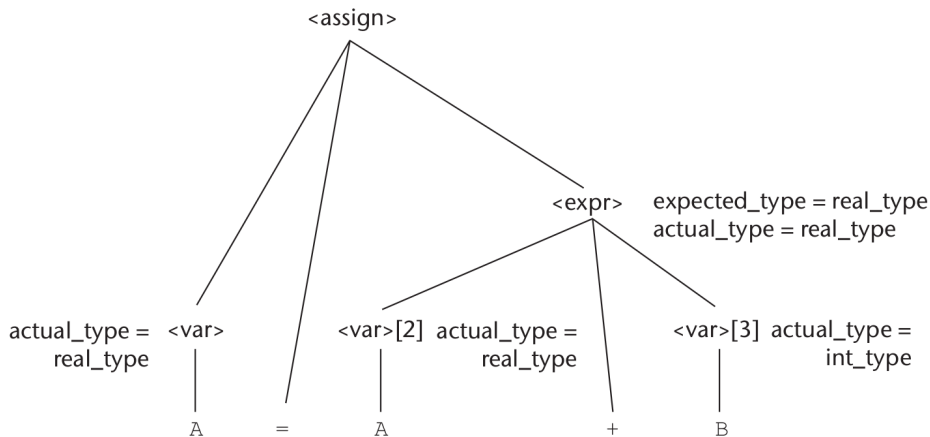
- برای محاسبه صفت‌ها در درخت تجزیه برای عبارت  $A = A + B$  داریم :

1.  $\langle \text{var} \rangle$ . actual-type  $\leftarrow$  look-up (A) (Rule 4)
2.  $\langle \text{expr} \rangle$ . expected-type  $\leftarrow$   $\langle \text{var} \rangle$ . actual-type (Rule 1)
3.  $\langle \text{var} \rangle [2]$ . actual-type  $\leftarrow$  look-up (A) (Rule 4)
- $\langle \text{var} \rangle [3]$ . actual-type  $\leftarrow$  look-up (B) (Rule 4)
4.  $\langle \text{expr} \rangle$ . actual-type  $\leftarrow$  either int or real (Rule 2)
5.  $\langle \text{expr} \rangle$ . expected-type =  $\langle \text{expr} \rangle$ . actual-type is either true or false (Rule 2)



## مثال گرامر صفت

- با فرض اینکه متغیر A از نوع real و متغیر B از نوع int باشد، درخت نهایی به صورت زیر خواهد بود.



- حال می‌خواهیم در مورد توصیف معنای برنامه صحبت کنیم، که به آن معناشناسی پویا<sup>1</sup> گفته می‌شود.
- از این پس منظور، از کلمه معناشناسی همان معناشناسی پویا است.
- توصیف معنای برنامه، به کاربران کمک خواهد کرد که معانی برنامه‌ها را بهتر متوجه شوند و همچنین به توسعه دهندگان کامپایلر کمک می‌کند تا بتوانند کامپایلر را به درستی پیاده سازی کنند و ابهامات و ناسازگاری‌های ممکن را رفع نمایند.
- اگر توصیف کاملی از نحو و معنای برنامه وجود داشته باشد، آنگاه می‌توانیم ابزاری تولید کنیم که به طور خودکار کامپایلر تولید کند.
- معمولاً برای توصیف معنا در یک زبان از زبان انگلیسی استفاده می‌شود که به دلیل غیر دقیق بودن، معمولاً یک توسعه دهنده کامپایلر باید در بسیاری موارد با آزمون و خطا یک کامپایلر را توسعه دهد و معمولاً برای زبان‌های رایج توصیف دقیقی وجود ندارد تا بتوان به طور خودکار کامپایلر آن را تهیه کرد. یکی از زبان‌هایی که برای معنای آن توصیف دقیقی داده شده است زبان اسکیم<sup>2</sup> است.

---

<sup>1</sup> dynamic semantic

<sup>2</sup> Scheme

- در معناشناسی عملیاتی<sup>1</sup> معنای عبارات یک برنامه با استفاده از تأثیر اجرای آنها بر روی ماشین توصیف می‌شود.
- تأثیر بر روی ماشین به معنی دنباله‌ای از تغییرات بر روی حالت ماشین است و حالت ماشین مجموعه‌ای از مقادیر بر روی حافظه آن است.

---

<sup>1</sup> operational semantics



- اولین گام در ساختن معناشناسی عملیاتی ساختن زبانی میانی است که برای توصیف به کار رود. مهم‌ترین معیاری که برای این زبان باید در نظر گرفته شود وضوح<sup>1</sup> آن است.
- هر ساختاری در این زبان باید روشن و غیر مبهم باشد. نیاز به چنین زبانی به این دلیل است که زبان ماشین بسیار پیچیده و ناخوانا است و زبان مورد نظر برای توصیف ناشناخته است.

---

<sup>1</sup> clarity

- برای مثال حلقه for در زبان سی را که به صورت زیر نوشته می شود.

---

```
۱  for ( expr1 ; expr2 ; expr3 ) { ... }
```

---

- می توانیم به صورت زیر توصیف کنیم.

---

```
۱          expr1;  
۲ loop :   if expr2 == 0 goto out;  
۳          ...  
۴          expr3;  
۵          goto loop;  
۶ out :    ...
```

---

- در چنین زبانی معمولاً از ساختارهای ساده‌ی زیر استفاده می‌کنیم.

```
۱ id = var  
۲ id = id + 1  
۳ id = id - 1  
۴ goto label  
۵ if var rel-op var goto label
```

- در اینجا از عملگرهای رابطه‌ی ای<sup>۱</sup> مانند = , > , < , >= , <= , != استفاده می‌کنیم.

- می‌توانیم این زبان را تعمیم دهیم و از عملگرهای حسابی ساده ریاضی مانند جمع و تفریق و ضرب و تقسیم و همچنین عملگرهای منطقی مانند و فصل و عطف و نقیض نیز استفاده کنیم.

- در معناشناسی عملیاتی برای توصیف یک زبان برنامه نویسی از یک زبان برنامه نویسی دیگر استفاده می‌کنیم. خواهیم دید که در روش‌های دیگر برای توصیف زبان می‌توانیم از زبان ریاضی استفاده کنیم.

---

<sup>1</sup> relational operator

- معناشناسی دلالتی<sup>1</sup> دقیق‌ترین و معروف‌ترین روش رسمی برای توصیف معنای برنامه‌هاست.
- این روش بر مبنای نظریه توابع گشتی است.
- توصیف معناشناسی دلالتی به طور کامل بسیار زمان بر است بنابراین در اینجا به قسمتی از آن و تعدادی مثال بسنده می‌کنیم.
- در توصیف معنای برنامه توسط معناشناسی دلالتی باید برای هر یک از ساختارهای برنامه یک ساختار ریاضی تعریف شود و توابعی تعریف شود که ساختارهای برنامه به ساختارهای ریاضی نگاشت کنند.
- به این روش معناشناسی دلالتی گفته می‌شود چرا که ساختارهای ریاضی دلیل استفاده مفاهیم در زبان را وصف می‌کنند.

---

<sup>1</sup> denotational semantics

- نگاشت ها در معناشناسی دلالتی مانند همه نگاشت ها یک دامنه و یک برد دارند. به دامنه این نگاشت ها دامنه نحوی<sup>1</sup> گفته می شود چرا که ساختارهای نحوی زبان را در بر می گیرند و به برد این نگاشت ها دامنه معنایی<sup>2</sup> گفته می شود.
- بنابراین در معناشناسی عملیاتی یک زبان را به یک زبان سطح پایین تر ترجمه می کنیم و در معناشناسی دلالتی زبان را به ساختارهای ریاضی ترجمه می کنیم.

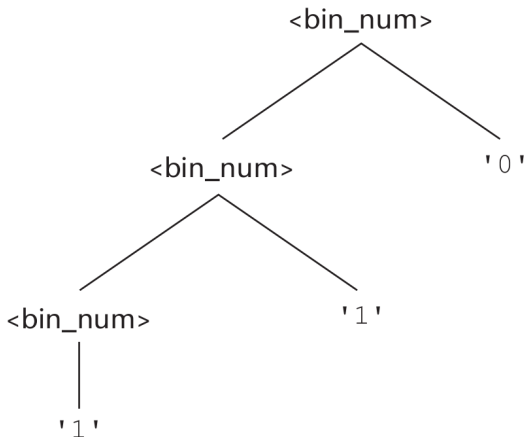
---

<sup>1</sup> syntactic domain

<sup>2</sup> semantic domain

- در اینجا یک قسمت بسیار ساده از یک زبان را در نظر می‌گیریم و آن را توسط معناشناسی دلالتی توصیف می‌کنیم.
- گرامری را در نظر بگیرید که یک عدد دودویی را به صورت رشته تولید می‌کند.  
$$\langle \text{bin-num} \rangle \rightarrow '0' \mid '1' \mid \langle \text{bin-num} \rangle '0' \mid \langle \text{bin-num} \rangle '1'$$

- درخت تجزیه برای جمله 110 از این گرامر در شکل زیر نشان داده شده است.



- دامنه نحوی در نگاشت معناشناسی دلالتی مجموعه همهٔ رشته‌های دودویی است. دامنه معنایی در این نگاشت مجموعه همه اعداد صحیح مثبت است که با  $N$  نمایش داده می‌شوند.
- اگر تابعی را تعریف کنیم که به ازای هر یک از قوانین گرامر با استفاده از مفهوم سمت چپ قانون معنی مفهوم سمت راست قانون را بیان کند، آنگاه می‌توانیم معنی متناظر با همه جملات زبان را به دست آوریم. در اینجا ساختار ریاضی که برای معنی جملات زبان به کار می‌بریم اعداد هستند.
- تابع نگاشت  $M_{bin}$  یک ساختار نحوی را با استفاده از قوانین گرامر به یک عدد نگاشت می‌کند.

$$M_{bin} ('0') = 0$$

$$M_{bin} ('1') = 1$$

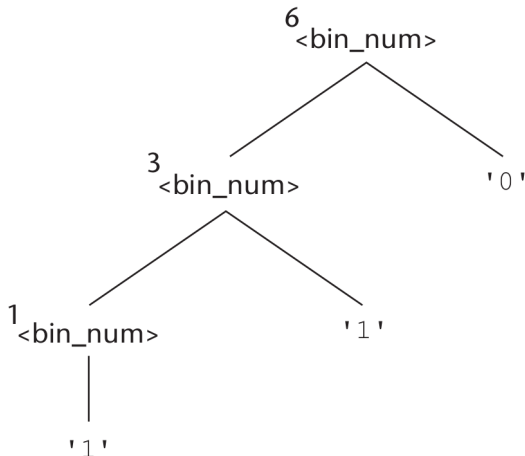
$$M_{bin} (<bin-num> '0') = 2 * M_{bin} (<bin-num> )$$

$$M_{bin} (<bin-num> '1') = 2 * M_{bin} (<bin-num> ) + 1$$



## معناشناسی دلالتی

- با استفاده از این معناشناسی دلالتی، می‌توانیم معنای همه رؤوس درخت تجزیه برای جمله '110' را تعیین کنیم.



- به طور مشابه می‌توانیم معنای گرامری که اعداد صحیح بدهی تولید می‌کند را با استفاده از معناشناسی دلالتی توصیف کنیم.

$$\langle \text{dec-num} \rangle \rightarrow '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9'$$

$$\mid \langle \text{dec-num} \rangle ( '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9' )$$

- معنای این گرامر به روش معناشناسی دلالتی به صورت زیر است.

$$M_{\text{dec}} ('0') = 0, M_{\text{dec}} ('1') = 1 \dots, M_{\text{dec}} ('9') = 9$$

$$M_{\text{dec}} (\text{dec-num}'0') = 10 * M_{\text{dec}} (\langle \text{dec-num} \rangle)$$

$$M_{\text{dec}} (\text{dec-num}'1') = 10 * M_{\text{dec}} (\langle \text{dec-num} \rangle) + 1$$

...

$$M_{\text{dec}} (\text{dec-num}'9') = 10 * M_{\text{dec}} (\langle \text{dec-num} \rangle) + 9$$

- در معناشناسی دلالتی، همانند معناشناسی عملیاتی، از توصیف تغییر حالت برنامه استفاده می‌کنیم. به طور دقیق‌تر در معناشناسی دلالتی از تغییر مقادیر متغیرها به زبان ریاضی استفاده می‌کنیم.
- فرض کنید حالت یک برنامه مقادیر متغیرهای آن باشد. به عبارت دیگر داشته باشیم :
$$S = \{ \langle i_1, V_1 \rangle, \langle i_2, V_2 \rangle, \dots, \langle i_n, V_n \rangle \}$$
- هر کدام از  $i$  ها یکی از متغیرها هستند که مقدار آن توسط  $V$  متناظر با آن نشان داده شده است. اگر متغیری مقدار نداشته باشد می‌توانیم از مقدار `undef` استفاده کنیم.
- فرض کنید تابع `VARMAP` مقدار یک متغیر در یک حالت دلخواه را باز می‌گرداند. به عبارت دیگر  $\text{VARMAP}(i_j, s)$  برابر است با  $V_j$ .

- حال فرض کنید گرامری برای عبارات محاسباتی در یک زبان برنامه نویسی داشته باشیم.

$\langle \text{expr} \rangle \rightarrow \langle \text{dec-num} \rangle \mid \langle \text{var} \rangle \mid \langle \text{binary-expr} \rangle$

$\langle \text{binary-expr} \rangle \rightarrow \langle \text{left-expr} \rangle \langle \text{operator} \rangle \langle \text{right-expr} \rangle$

$\langle \text{left-expr} \rangle \rightarrow \langle \text{dec-num} \rangle \mid \langle \text{var} \rangle$

$\langle \text{right-expr} \rangle \rightarrow \langle \text{dec-num} \rangle \mid \langle \text{var} \rangle$

$\langle \text{operator} \rangle \rightarrow + \mid *$

- تنها خطایی که می‌تواند وجود داشته باشد این است که مقداری تعریف نشده باشد. اگر مجموعه همه اعداد صحیح را  $Z$  و مقدار خطا را برابر با  $\text{error}$  در نظر بگیریم، آنگاه  $Z \cup \{\text{error}\}$  دامنه معنایی برای عبارات این زبان است.

- در اینجا برای عملگر تساوی ریاضی از علامت  $\Delta$  استفاده می‌کنیم، چرا که علامت  $=$  معمولاً برای انتساب مقدار در زبان برنامه نویسی استفاده می‌شود. همچنین علامت  $\Rightarrow$  به معنی بازگرداندن مقدار یا نتیجه عبارت است.

$$\begin{aligned}
M_e(\langle \text{expr} \rangle, s) \Delta = & \text{case } \langle \text{expr} \rangle \text{ of} \\
& \langle \text{dec-num} \rangle \Rightarrow M_{\text{dec}}(\langle \text{dec-num} \rangle, s) \\
& \langle \text{var} \rangle \Rightarrow \text{if VARMAP}(\langle \text{var} \rangle, s) = \text{undef} \\
& \quad \text{then error} \\
& \quad \text{else VARMAP}(\langle \text{var} \rangle, s) \\
& \langle \text{binary-expr} \rangle \Rightarrow \\
& \quad \text{if } (M_e(\langle \text{binary-expr} \rangle.\langle \text{left-expr} \rangle, s) = \text{undef OR} \\
& \quad \quad M_e(\langle \text{binary-expr} \rangle.\langle \text{right-expr} \rangle, s) = \text{undef}) \\
& \quad \text{then error} \\
& \quad \text{else if } (\langle \text{binary-expr} \rangle.\langle \text{operator} \rangle = '+' ) \\
& \quad \quad \text{then } M_e(\langle \text{binary-expr} \rangle.\langle \text{left-expr} \rangle, s) + \\
& \quad \quad \quad M_e(\langle \text{binary-expr} \rangle.\langle \text{right-expr} \rangle, s) \\
& \quad \quad \text{else } M_e(\langle \text{binary-expr} \rangle.\langle \text{left-expr} \rangle, s) * \\
& \quad \quad \quad M_e(\langle \text{binary-expr} \rangle.\langle \text{right-expr} \rangle, s)
\end{aligned}$$

- مقدار محاسبه شده برای  $M_e$  مقدار یک عبارت را محاسبه می‌کند و اگر بخواهیم معنی یک عبارت انتساب را بیان کنیم باید حالت جدید ماشین را محاسبه کنیم.

$M_a (x = E , s) \Delta =$  if  $M_e (E , s) == \text{error}$   
 then error  
 else  $s' = \{ \langle i_1 , v'_1 \rangle , \langle i_2 , v'_2 \rangle , \dots , \langle i_n , v'_n \rangle \}$  , where  
 for  $j = 1, 2, \dots , n$   
 if  $i_j = x$   
 then  $v'_j = M_e (E , s)$   
 else  $v'_j = \text{VARMAP}(i_j, s)$

- دقت کنید که  $i_j = x$  مقایسه اسامی است نه مقایسه مقادیر.

- تلاش‌های بسیاری برای تعریف معنای زبان‌های برنامه نویسی توسط معناشناسی دلالتی صورت گرفته است. برای اینکه یک کامپایلر از روی تعریف نحو و معنا به طور خودکار ساخته شود اما این تلاش‌ها به نتیجه‌ای نرسیده است. با این وجود تعریف زبان به وسیله معناشناسی دلالتی تعریف دقیقی از زبان به دست می‌دهد و همچنین پیچیده شدن بیش از حد تعاریف نشان از این خواهد بود که زبان برنامه نویسی مورد توصیف برای کاربران نیز پیچیده خواهد شد.



- معناشناسی اصل موضوعی<sup>1</sup> به این نام خوانده می‌شود چرا که بر مبنای منطق ریاضی<sup>2</sup> و استنتاج گزاره‌ها بر اساس اصول موضوع<sup>3</sup> است.
- در معناشناسی اصل موضوعی معنی یک برنامه بر اساس ارتباط متغیرها و ثابت‌های آن مشخص می‌شود.

---

<sup>1</sup> Axiomatic semantics

<sup>2</sup> mathematical logic

<sup>3</sup> axioms

- در معناشناسی اصل موضوعی به ازای عبارت  $S$  ، یک پیش شرط  $P$  و یک پس شرط  $Q$  می نویسیم.  
 $\{P\} S \{Q\}$
- اگر بخواهیم اثبات کنیم یک برنامه درست است، باید اثبات کنیم به ازای مجموعه ای از ورودی ها خروجی های مناسب تولید می شوند.
- اگر به ازای پیش شرط یک برنامه پس شرط برنامه در محدوده مقادیر مورد انتظار نباشد، برنامه درست نیست.

- به عنوان مثال عبارت زیر را در نظر بگیرید.

$$x = x + y - 3 \{ x > 10 \}$$

- می‌توانیم پیش شرط را بدین صورت محاسبه کنیم :

$$x + y - 3 > 10 , y > 13 - x$$

- بنابراین می‌نویسیم :

$$\{ y > 13 - x \} x = x + y - 3 \{ x > 10 \}$$

## معناشناسی اصل موضوعی

- فرض کنید می‌خواهیم درستی برنامه زیر را اثبات کنیم.

$$\{ x = A \text{ AND } y=B \} \ t = x ; x = y ; y = t ; \{ x = B \text{ AND } y = A \}$$

- از آخرین دستور شروع می‌کنیم و پیش شرط دستور سوم را محاسبه می‌کنیم. این پیش شرط برابر است با

$$\{ x = B \text{ AND } t = A \}$$

- سپس پیش شرط دستور سوم را به عنوان پس شرط دستور دوم در نظر می‌گیریم. پیش شرط دستور دوم برابر خواهد بود با

$$\{ y = B \text{ AND } t = A \}$$

- سپس پیش شرط دستور دوم را به عنوان پس شرط دستور اول در نظر می‌گیریم. پیش شرط دستور اول برابر خواهد بود با

$$\{ y = B \text{ AND } x = A \}$$

- بنابراین این برنامه درست است.

- برای همه ساختارهای یک برنامه از جمله دستورات شرطی و حلقه‌ها می‌توانیم پیش‌شرط و پس‌شرط‌ها را بر اساس گزاره‌های منطقی محاسبه کنیم که به علت طویل بودن محاسبات در اینجا از آنها صرف نظر می‌کنیم.