به نام خدا

زبان‌های برنامه‌نویسی

آرش شفیعی

نحو و معناشناسی

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 [1] 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 [2] ، معناشناسی دلالتی [3] ، و معناشناسی اصلی موضوعی [4] را توضیح می‌دهیم.

[1] attribute grammar
[2] operational semantics
[3] denotational semantics
[4] axiomatic semantics

نحو و معناشناسی

- 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 استفاده کنندگان زبان نیاز دارند ویژگی‌های زبان را به درستی بشناسند.
- مطالعه زبان‌های برنامه نویسی مانند مطالعه زبان‌های طبیعی می‌تواند به دو قسمت تقسیم شود : نحو شناسی [1] به مطالعه صورت عبارت در زبان مورد نظر و معناشناسی [2] به مطالعه معانی عبارات می‌پردازد.
- به عبارت دیگر در نحو شناسی مطالعه می‌کنیم چگونه کلمات ترکیب می‌شوند تا عبارات و جمله‌ها را بسازند، اما در معناشناسی به مطالعهٔ معنی آن عبارات و جملات و درستی و نادرستی آنها می‌پردازیم.

[1] syntax
[2] semantics

نحو و معناشناسی

- 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.

توصیف نحوی

- 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.
- قوائد 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 قوائد نحوی است که جملات زبان را تعیین می‌کنند. در مقایسه با زبان انگلیسی، حتی پیچیده‌ترین و بزرگ‌ترین زبان‌های برنامه‌نویسی قوائد نحوی ساده‌تر و کمتری دارند.

توصیف نحوی

- توصیف رسمی قوائد نحوی زبان‌های برنامه‌نویسی، معمولاً نحوهٔ تولید کوچک‌ترین اجزای زبان را مشخص نمی‌کند. کوچکترین اجزای یک جمله را تکواژه [1] یا لغت می‌نامیم. توصیف لغات را می‌توان به طور جداگانه با استفاده از یک توصیف کنندهٔ 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 [2] 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 [3] می‌گوییم.

[1] lexeme
[2] identifier
[3] token

توصیف نحوی

- عبارت `index = 2 * 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

توصیف نحوی

- زبان‌ها را می‌توان به دو روش توصیف کرد. به وسیله تشخیص [1] یا به وسیله تولید [2] .

[1] recognition
[2] generation

تشخیص دهندهٔ زبان

- فرض کنید زبان L را داریم که از الفبای Σ استفاده می‌کند. برای تعریف زبان L توسط تشخیص دهنده باید از یک مکانیزم R به عنوان دستگاه تشخیص دهندهٔ زبان استفاده کنیم که قادر باشد رشته‌هایی که از الفبای Σ 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 همهٔ رشته‌ها بر روی الفبای Σ، 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.

تولید کننده زبان

- یک تولید کننده زبان دستگاهی است که جملات یک زبان را تولید می‌کند.
- مکانیزمی که توسط آن یک زبان تولید می‌شود گرامر [1] نامیده می‌شود.
- توسط یک دستگاه تولید کنندهٔ زبان یا یک گرامر می‌توانیم بررسی کنیم آیا یک برنامه توسط آن گرامر قابل تولید است یا خیر.
- یکی از روش‌های توصیف قوائد نحوی، استفاده از گرامر است. معمولاً برای توصیف زبان از گرامر آن استفاده می‌کنیم.

[1] grammar

گرامرهای مستقل از متن

- در اواسط دههٔ ۱۹۵۰ نوام چامسکی [1] 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 [2] و گرامرهای مستقل از متن [3] 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.

[1] Noam Chamsky
[2] Regular
[3] Context-free

گرامرهای مستقل از متن

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

[1] John Backus
[2] Peter Naur
[3] Backus-Naur form (BNF)

گرامرهای مستقل از متن

- یک فرا زبان [1] 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 [2] در زبان جاوا را می‌توانیم با مفهوم <assign> نمایش دهیم. می‌توانیم بنویسیم : <assign> → <var> = <expression>

- عبارت سمت چپ علامت فلش که lhs [1] نامیده می‌شود، مفهومی است که توسط این گرامر می‌خواهیم تعریف کنیم. عبارت سمت راست علامت فلش که rhs [2] نامیده می‌شود، تشکیل شده‌است از ترکیبی از توکن‌ها و مفاهیم دیگر. به طور کلی عبارت lhs → rhs را یک قانون تولید [3] می‌نامیم.

---

[1] metalanguage
[2] assignment expression
[1] left-hand side
[2] right-hand side
[3] production rule

گرامرهای مستقل از متن

- در مثالی که بیان شد مفاهیم <var> و <expression> باید تعریف شوند تا <assign> بتواند کاملا 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 [1] یا متغیر و توکن‌ها را نمادهای پایانی [2] یا ترمینال می‌نامیم. یک گرامر از تعدادی قوانین تولید تشکیل شده‌است.

[1] nonterminal symbols
[2] terminal symbols

گرامرهای مستقل از متن

- 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.

<if-stmt> → if (<logic-expr>) <stmt>
<if-stmt> → if (<logic-expr>) <stmt> else <stmt>

- برای ترکیب دو قانون بالا می‌توانیم بنویسیم :

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

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 می تواند 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.

اشتقاق

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

[1] start symbol
[2] derivation

اشتقاق

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

<program> → begin <stmt-list> end
<stmt-list> → <stmt> ; | <stmt> ; <stmt-list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var> | <var> - <var> | <var>

اشتقاق

- یک فرایند اشتقاق به صورت زیر است.

<program> ⇒ begin <stmt-list> end
⇒ begin <stmt> ; <stmt-list> end
⇒ begin <var> = <expresssion> ; <stmt-list> end
...
⇒ begin A = B + C ; B = C ; end

- علامت ⇒ را می‌خوانیم «به دست می‌دهد» یا «می‌دهد» یا «مشتق می‌کند».

اشتقاق

- 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 [1] می‌نامیم.
- اگر در فرایند اشتقاق همیشه ابتدا اولین متغیر از سمت چپ جایگزین شود، آن فرایند را اشتقاق چپ [2] 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.

[1] sentential form
[2] leftmost derivation

اشتقاق

- 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.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr> | <id> * <expr> | (<expr>) | <id>

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

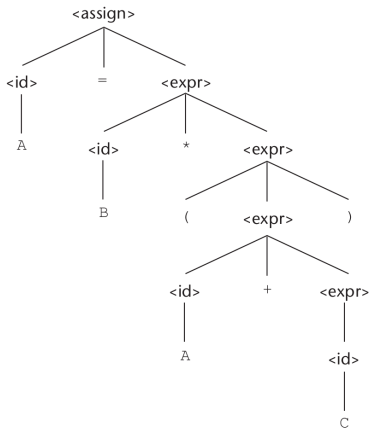
درخت تجزیه

- 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 [1] نامیده می‌شود.

[1] 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.



درخت تجزیه

- 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.

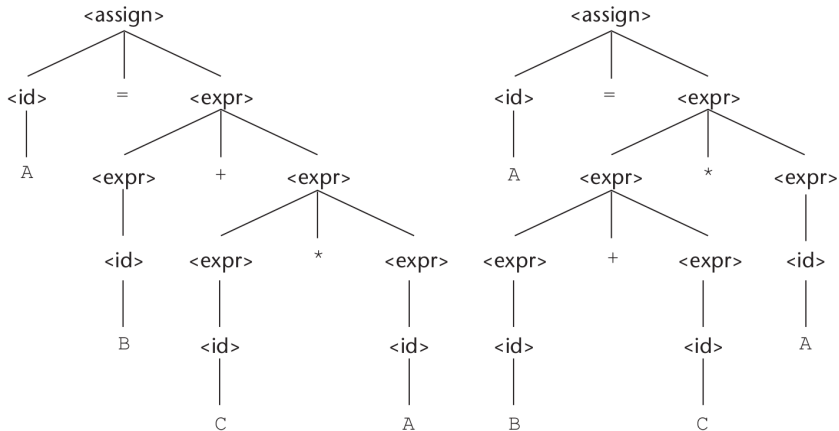
ابهام

- 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 زیر برای تولید عبارات تخصیص مقدار را در نظر بگیرید.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr> | <expr> ∗ <expr> | (<expr>) | <id>

ابهام

- برای جمله `A = B + C * A` دو درخت تجزیه متفاوت وجود دارد.



ابهام

- 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.
- 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 عملگرجمع 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.

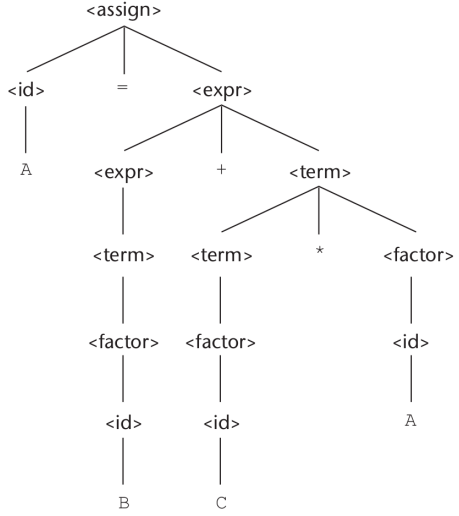
تقدم عملگرها

- گرامر غیر مبهم زیر معادل گرامر مبهم قبلی است.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → (<expr>) | <id>

تقدم عملگرها

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



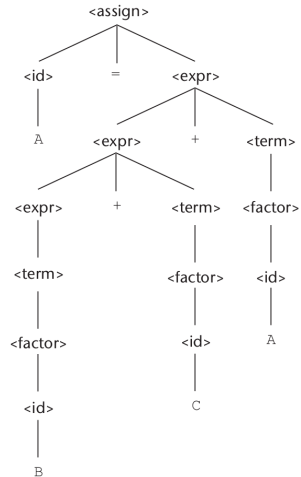
وابستگی عملگرها

- 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 [1] مشخص می‌کند که در شرایطی که تقدم یکسان است به کدام عملگر اولویت بالاتری داده می‌شود.

[1] 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 [1] 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 1.0000000×10^7 خواهد بود.
- اما اگر ۱۰ عدد ۱ را با هم جمع کنیم و در نهایت با 10^7 جمع کنیم نتیجه 1.0000001×10^7 خواهد شد.

[1] associative property

وابستگی عملگرها

- همچنین اگر عملگرهای تفریق داشته باشیم، نتیجه عملیات $1 - 2 - 5$ بسته به این که از چپ به راست یا راست به چپ محاسبه شود، متفاوت خواهد بود.
- اگر وابستگی چپ داشته باشیم، نتیجه عبارت برابر است با $(1 - 2) - 5 = -6$ و اگر وابستگی راست داشته باشیم، نتیجه عبارت برابر است با $1 - (2 - 5) = 4$.
- وابستگی عملگر تفریق از چپ به راست است، و نتیجه مورد نظر برابر است با -6 .

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 اشاره گر را ارزیابی و سپس به آن یک واحد بیافزاییم، از عبارت ++(p*) استفاده می‌کنیم.

وابستگی عملگرها

- وقتی در یک گرامر، یک مفهوم یا متغیر lhs در یک قانون، در طرف چپ rhs باشد می‌گوییم این قانون بازگشتی چپ [1] 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 [2] است. یک قانون بازگشتی راست تولید وابستگی از راست می‌کند.

<factor> → id ∗∗ <factor> | id

[1] left recursive
[2] right recursive

گرامرهای غیر مبهم برای if-else

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

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

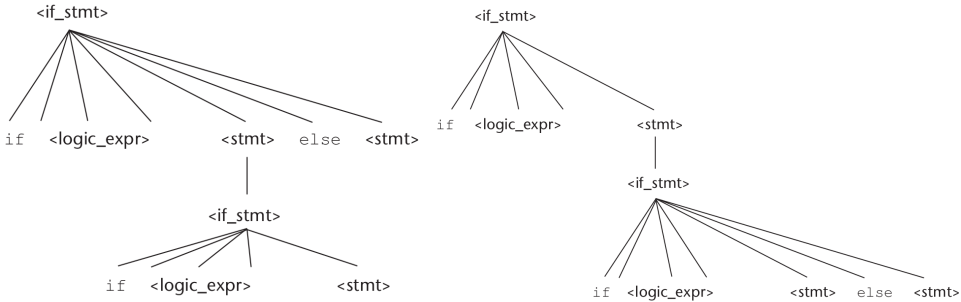
- حال فرض کنید می‌خواهیم if های تو در تو داشته باشیم. اگر قانون <if-stmt> → <stmt> را اضافه کنیم، این گرامر تبدیل به یک گرامر مبهم می‌شود.
- یکی از صورت‌های جمله‌ای که توسط این گرامر ساخته می‌شود برابر است با

if(<logic-expr>) if(<logic-expr>) <stmt> else <stmt>

- 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 <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

<id-list> → <id> { , <id> }

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

<term> → <term> (* | / | %) <factor>

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

<term> → <term> * <factor> | <term> / <factor> | <term> % <factor>

- 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.

<expr> → <term> { (+ | -) <term> }
<term> → <factor> { (* | /) <factor> }
<factor> → <exp> { ** <exp> }
<exp> → (<expr>) | <id>

گرامرهای مستقل از متن تعمیم یافته

- قبلا 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.

گرامر صفت

- یک گرامر صفت [1] 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.

[1] attribute grammar

گرامر صفت

- برخی از ویژگی‌های زبان‌های برنامه‌نویسی نمی‌توانند توسط گرامرهای مستقل از متن بیان شوند. یکی از این ویژگی‌ها سازگاری نوع داده‌ها [1] است.
- برای مثال در زبان جاوا متغیرهای نوع اعشاری 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 خواهد شد.

[1] type compatibility

گرامر صفت

- 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 [1] می‌گوییم.
- در واقع قوانین معناشناسی ایستا مربوط به قوانین معنایی برنامه هستند که در زمان کامپایل قابل بررسی هستند.

[1] static semantic rules

گرامر صفت

- 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 تقریبا در همهٔ کامپایلرها به صورت غیر رسمی استفاده شده‌اند.
- معناشناسی پویا مربوط به معانی عبارات در زمان اجرا است که بعدها به آن اشاره خواهیم کرد.

گرامر صفت

- گرامرهای صفت در واقع گرامرهای مستقل از متن هستند که به آنها تعدادی صفت، توابع محاسبه صفت [1] و توابع مسندی [2] 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.

[1] attribute computation function
[2] predicate function

تعریف گرامر صفت

- به ازای هر نماد X در یک گرامر مجموعه‌ای از صفت‌ها به نام A(X) وجود دارد. مجموعهٔ A(X) به دو مجموعهٔ مجزا افراز می‌شود. مجموعه‌های S(X) و I(X) که صفت‌های ترکیبی [1] و صفت‌های موروثی [2] نامیده می‌شوند.

[1] Synthesized attributes
[2] Inherited attributes

تعریف گرامر صفت

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

$X_0 \to X_1 \cdots X_n$

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

- صفت‌های موروثی به صورت $I(X_j) = f(A(X_0), \cdots, A(X_{j-1}), A(X_{j+1}), \cdots, A(X_n))$ محاسبه می‌شوند. پس صفت‌های موروثی به صفت‌های پدر و همزادها [3] بستگی دارد.
- صفت‌های ترکیبی معانی را در درخت تجزیه به بالا منتقل می‌کنند در حالی که صفت‌های موروثی معانی را در درخت تجزیه به پایین منتقل می‌کنند.

[3] sibiling

تعریف گرامر صفت

- 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.

تعریف گرامر صفت

- صفت های ذاتی [1] صفت‌های ترکیبی مربوط به برگ‌های درخت تجزیه هستند. برای مثال نوع یک متغیر در یک برنامه یک صفت ذاتی است که می‌توان آن را از جدول نمادها دریافت کرد. هنگامی که یک درخت تجزیه ساخته می‌شود، اولین ویژگی‌هایی که قابل محاسبه هستند، ویژگی‌های ذاتی برگ‌های درخت هستند. پس از آن، با استفاده از توابع صفت می‌توان صفت‌های رئوس دیگر را محاسبه نمود.

[1] intrinsic attributes

مثال گرامر صفت

- برای مثال در زبان آدا [1] ، 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 : <proc-def> → procedure <proc-name>[1]
<proc-body> end <proc-name>[2]
Predicate : <proc-name>[1] string == <proc-name>[2] 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 کلمهٔ کلیدی procedure است، باید با نام تابع که انتهای تعریف تابع، بعد از کلمهٔ کلیدی end نوشته می‌شود، همخوانی داشته باشد.

[1] 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 :

<assign> → <var> = <expr>
<expr> → <var> + <var> | <var>
<var> → A | B | C

مثال گرامر صفت

- 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 ترکیبی 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 موروثی 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

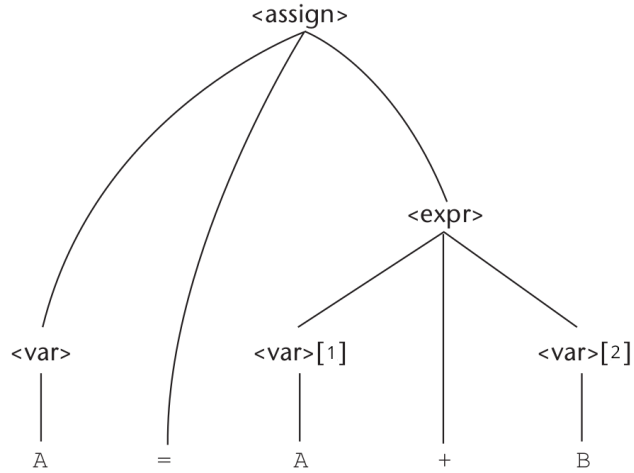
- گرامر صفت برای مثال قبلی را به صورت زیر می‌نویسیم.

1. Syntax rule : <assign> → <var> = <expr>
Semantic rule : <expr> . expected-type ← <var> . actual-type
2. Syntax rule : <expr> → <var> [1]+ <var> [2]
Semantic rule : <expr> . actual-type ← if (<var> [1] . actual-type = int)
and (<var> [2] . actual-type = int)
then int
else real
and if
Predicte : <expr> . actual-type == <expr> . expected-type
3. Syntax rule : <expr> → <var>
Semantic rule : <expr> . actual-type ← <var> . actual-type
Predicte : <expr> . actual-type == <expr> . expected-type
4. Syntax rule : <var> → A | B | C
Semantic rule : <var> . actual-type ← look-up (<var> . 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.

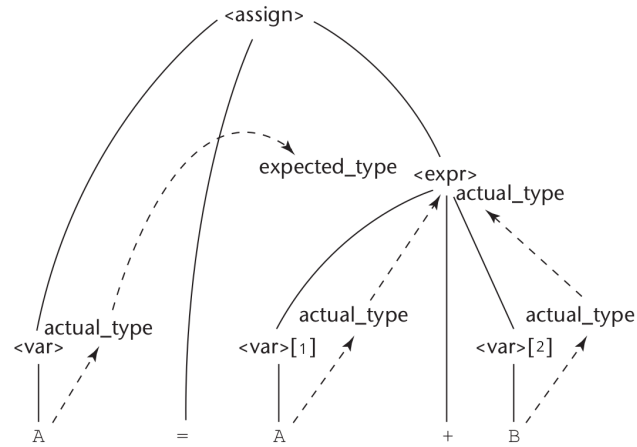


مثال گرامر صفت

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

مثال گرامر صفت

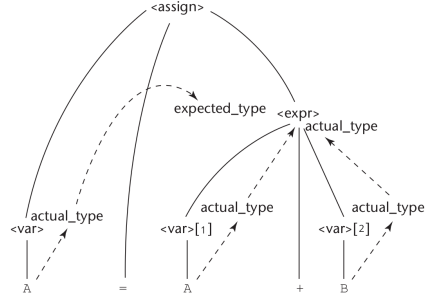
- شکل زیر نحوهٔ محاسبه صفت‌ها در درخت تجزیه را نشان می‌دهد. نوع واقعی (actual-type) یک صفت ترکیبی است، درحالی‌که نوع مورد انتظار (expected-type) یک صفت موروثی است.



مثال گرامر صفت

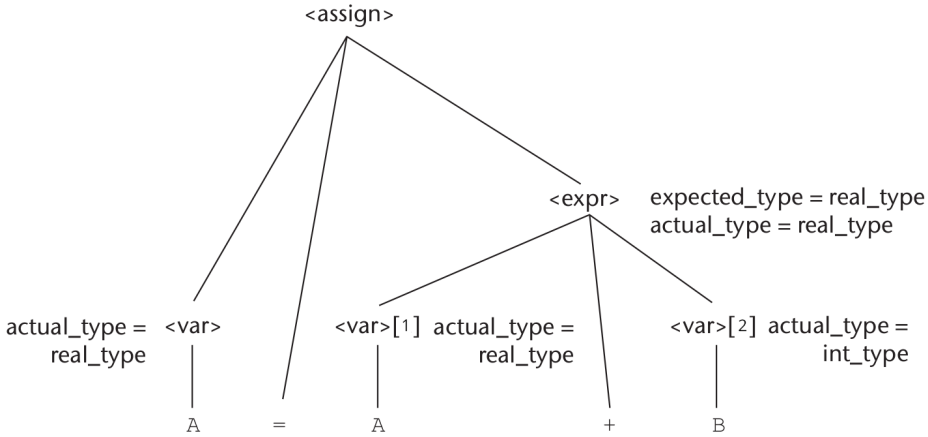
- برای محاسبه صفت‌ها در درخت تجزیه برای عبارت A = A + B داریم :

1. <var>. actual-type ← look-up (A) (Rule 4)
2. <expr>. expected-type ← <var>. actual-type (Rule 1)
3. <var> [1]. actual-type ← look-up (A) (Rule 4)
<var> [2]. actual-type ← look-up (B) (Rule 4)
4. <expr>. actual-type ← either int or real (Rule 2)
5. <expr>. expected-type == <expr>. 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.



معناشناسی پویا

- گرامر صفت روشی است که برای معناشناسی ایستا (تحلیل معنایی برنامه) به کار می‌رود. در معناشناسی ایستا بررسی می‌کنیم آیا برنامه از نظر معنایی درست است یا خیر. برای مثال آیا متغیرها قبل از استفاده تعریف شده‌اند یا خیر. معناشناسی ایستا در واقع تعمیمی برای تحلیل نحوی برنامه است.
- در معناشناسی پویا [1] می‌خواهیم معنای یک برنامه یا به عبارت دیگر معنای قوانین گرامر یک زبان را توصیف کنیم. برای توصیف معنای برنامه باید از زبانی دیگر استفاده کنیم. برای مثال می‌توانیم از یک زبان سطح پایین تر یا از زبان ریاضی برای توصیف معنای یک برنامه استفاده می‌کنیم.
- سه روش مهم برای معناشناسی پویا وجود دارد: معناشناسی عملیاتی، معناشناسی دلالتی، و معناشناسی اصل موضوعی.

[1] dynamic semantic

معناشناسی پویا

- معناشناسی پویا کاربردهای متنوعی دارد.
- توصیف دقیق یک زبان و توصیف معنای برنامه‌های آن می‌تواند توسط معناشناسی پویا انجام شود. برای مثال در معناشناسی دلالتی از زبان ریاضی برای توصیف دقیق یک زبان استفاده می‌شود.
- اثبات درستی برنامه‌ها نیز می‌تواند توسط معناشناسی پویا انجام گیرد. برای مثال با استفاده از معناشناسی اصل موضوعی می‌توانیم اثبات کنیم یک برنامه خروجی درست تولید می‌کند.
- تولید خودکار تولیدکننده کد در کامپایلر از دیگر کاربردهای معناشناسی پویا است. برای مثال با استفاده از معناشناسی عملیاتی می‌توانیم قوانین یک گرامر را با یک زبان سطح پایین توصیف کنیم و از این توصیف برای تولید خودکار تولیدکننده کد استفاده کنیم.

معناشناسی پویا

- 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 ومعنای برنامه وجود داشته باشد، آنگاه می‌توانیم ابزاری تولید کنیم که به طور خودکار کامپایلر تولید کنیم.

- معمولا برای توصیف معنا در یک زبان از زبان انگلیسی استفاده می‌شود که به دلیل غیر دقیق بودن، معمولاً یک توسعه دهنده کامپایلر باید 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 [2] است.

[2] Scheme

معناشناسی عملیاتی

- در معناشناسی عملیاتی [1] 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 حافظهٔ آن است.

[1] operational semantics

معناشناسی عملیاتی

- 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 [1] 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.

[1] clarity

معناشناسی عملیاتی

- برای مثال حلقه for در زبان سی را که به صورت زیر نوشته می‌شود،

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

می‌توانیم به صورت زیر توصیف کنیم.

```
          expr1;
loop :    if expr2 == 0 goto out;
          ...
          expr3;
          goto loop;
out  :    ...
```

معناشناسی عملیاتی

- در چنین زبانی معمولا از ساختارهای سادهٔ 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 رابطه ای [1] 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 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.

[1] relational operator

معناشناسی دلالتی

- معناشناسی دلالتی [1] دقیق‌ترین روش رسمی برای توصیف معنای برنامه‌هاست. در این روش برای توصیف زبان، از زبان ریاضی استفاده می‌کنیم.
- توصیف معناشناسی دلالتی به طور کامل بسیار زمان بر است، بنابراین در اینجا به توضیح 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 را به ساختارهای ریاضی نگاشت کنند.
- به این روش معناشناسی دلالتی گفته می‌شود چرا که ساختارهای ریاضی دلیل استفاده مفاهیم در زبان را وصف می‌کنند.

[1] denotational semantics

معناشناسی دلالتی

- نگاشت ها در معناشناسی دلالتی مانند همهٔ نگاشت‌ها یک دامنه و یک برد دارند. به دامنهٔ این نگاشت‌ها دامنه نحوی [1] گفته می‌شود چرا که ساختارهای نحوی زبان را در بر می‌گیرند و به برد این نگاشت‌ها دامنه معنایی [2] گفته می‌شود.
- بنابراین در معناشناسی عملیاتی یک زبان را به یک زبان سطح پایین‌تر ترجمه می‌کنیم و در معناشناسی دلالتی زبان را به ساختارهای ریاضی ترجمه می‌کنیم.

[1] syntactic domain
[2] semantic domain

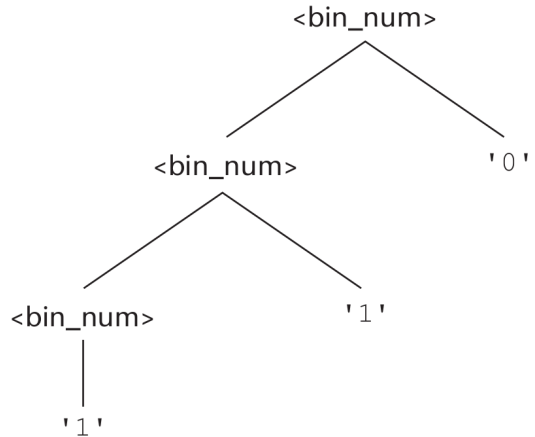
معناشناسی دلالتی

- 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.

<bin-num> → '0' | '1' | <bin-num> '0' | <bin-num> '1'

معناشناسی دلالتی

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



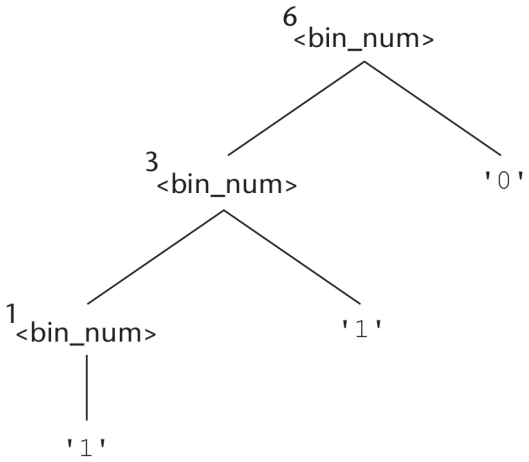
معناشناسی دلالتی

- دامنه نحوی در نگاشت معناشناسی دلالتی مجموعه همهٔ رشته‌های دودویی است. دامنه معنایی در این نگاشت مجموعه همه اعداد حسابی است.
- 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 رئوس درخت تجزیه برای جمله '110' را تعیین کنیم.



معناشناسی دلالتی

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

<dec-num> → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
| <dec-num> ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')

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

M_{dec} ('0') = 0 , M_{dec} ('1') = 1 ⋯ , M_{dec} ('9') = 9
M_{dec} (dec-num'0') = 10 * M_{dec} (<dec-num>)
M_{dec} (dec-num'1') = 10 * M_{dec} (<dec-num>) + 1
...
M_{dec} (dec-num'9') = 10 * M_{dec} (<dec-num>) + 9

معناشناسی دلالتی

- 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 :

$$s = \{ (x_1 , v_1) , (x_2 , v_2) , \cdots , (x_n , v_n) \}$$

- هر کدام از x 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 VARMAP (x_j , s) برابر است با v_j.

معناشناسی دلالتی

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

<expr> $\rightarrow$ <dec-num> | <var> | <binary-expr>
<binary-expr> $\rightarrow$ <left-expr> <operator> <right-expr>
<left-expr> $\rightarrow$ <dec-num> | <var>
<right-expr> $\rightarrow$ <dec-num> | <var>
<operator> $\rightarrow$ + | *

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

معناشناسی دلالتی

- در اینجا برای عملگر تساوی ریاضی از علامت $=\Delta$ استفاده می‌کنیم، چرا که علامت = معمولا برای انتساب مقدار در زبان برنامه نویسی استفاده می‌شود. همچنین علامت $\Rightarrow$ به معنی بازگرداندن مقدار یا نتیجه عبارت است.
- مقدار محاسبه شده برای M_e مقدار یک عبارت در این زبان را محاسبه می‌کند.

معناشناسی دلالتی

M_e(<expr> , s) Δ = case <expr> of
<dec-num> $\Rightarrow$ M_{dec} (<dec-num> , s)
<var> $\Rightarrow$ if VARMAP (<var> , s) == undef
then error
else VARMAP (<var> , s)
<binary-expr> $\Rightarrow$
if (M_e (<binary-expr>.<left-expr>, s))== undef OR
M_e (<binary-expr>.<right-expr>, s) == undef
then error
else if (<binary-expr>.<operator> == '+')
then M_e (<binary-expr>.<left-expr> , s) +
M_e (<binary-expr>.<right-expr> , s)
else M_e (<binary-expr>.<left-expr> , s) *
M_e (<binary-expr>.<right-expr> , s)

معناشناسی دلالتی

- اگر بخواهیم معنی یک عبارت انتساب را بیان کنیم باید حالت جدید ماشین را محاسبه کنیم.

$$
\begin{aligned}
M_a\,(p = expr\ ,\ s)\ \Delta = {} & \text{if } M_e\,(expr\ ,\ s) == error \\
& \text{then error} \\
& \text{else } s' = \{\ (x_1\ ,\ v_1')\ ,\ (x_2\ ,\ v_2')\ ,\ \cdots\ ,\ (i_n\ ,\ v_n')\ \}\ ,\ \text{where} \\
& \quad \text{for } j = 1,\ 2,\ \cdots\ ,\ n \\
& \quad\ \text{if } x_j == p \\
& \quad\quad \text{then } v_j' = M_e\,(expr\ ,\ s) \\
& \quad\quad \text{else } v_j' = \text{VARMAP}(x_j,\ s)
\end{aligned}
$$

- دقت کنید که $x_j == p$ مقایسه اسامی است نه مقایسه مقادیر.

معناشناسی دلالتی

- 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

- معناشناسی اصل موضوعی [1] به این نام خوانده می‌شود چرا که بر مبنای منطق ریاضی [2] و استنتاج گزاره‌ها بر اساس اصول موضوع [3] است.
- در معناشناسی اصل موضوعی معنی یک برنامه بر اساس ارتباط متغیرهای آن با یکدیگر مشخص می‌شود.

[1] Axiomatic semantics
[2] mathematical logic
[3] axioms

معناشناسی اصل موضوعی

- در معناشناسی اصل موضوعی به ازای عبارت S ، یک پیش شرط P و یک پس شرط Q می‌نویسیم.

$$\{P\}\ S\ \{Q\}$$

- اگر بخواهیم اثبات کنیم یک برنامه درست است، باید اثبات کنیم به ازای مجموعه‌ای از ورودی‌ها خروجی‌های مناسب تولید می‌شوند.
- اگر به ازای پیش شرط یک برنامه پس شرط برنامه در محدودهٔ مقادیر مورد انتظار نباشد، برنامه درست نیست.

معناشناسی اصل موضوعی

- فرض کنید می‌خواهیم درستی برنامه زیر را اثبات کنیم.

```
{ x = A AND y=B } t = x ; x = y ; y = t ; { x = B AND y = A }
```

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

```
{ x = B 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 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 AND x = A }
```

- بنابراین این برنامه درست است.

معناشناسی اصل موضوعی

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