ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 2

Синтаксис

Весна 2021

Коли ви навчилися програмувати на одній мові, вивчити подібну мову програмування не так вже й складно. Але, щоб зрозуміти, як правильно писати новою мовою, потрібно переглянути приклади чи прочитати документацію, щоб дізнатись її деталі. Іншими словами, вам потрібно знати механіку складання програми новою мовою. Чи стоять крапки з комою в потрібних місцях? Чи використовуєте ви початок ... кінець чи ви використовуєте фігурні дужки (тобто {і})? Вивчення того, як складається програма, називається вивченням синтаксису мови. *Синтаксис* стосується слів та

символів мови та способу запису символів у певному значущому порядку.

Семантика - це слово, яке використовується при виведенні значення із написаного. Семантика програми стосується того, що програма буде робити при її виконанні. Неофіційно набагато простіше сказати, що робить програма, ніж описати синтаксичну структуру програми. Однак формально описати синтаксис набагато простіше, ніж семантику. У будь-якому випадку, якщо ви вивчаєте нову мову, вам потрібно дізнатися щось як про синтаксис, так і про семантику мови.

Ще раз, синтаксис мови програмування визначає правильно сформовані або граматично правильні програми мови. Семантика описує, як і чи будуть виконуватися такі програми.
Синтаксис - це те, як виглядають програми
Семантика - це те, як працюють програми
Багато питань, які ми можемо поставити про програму, стосуються або синтаксису мови, або її семантики. Не завжди зрозуміло, які питання стосуються синтаксису, а які - семантики.

Деякі питання можуть стосуватися семантичних питань, які можна визначити статично, тобто перед запуском програми.
Інші семантичні проблеми можуть бути динамічними, тобто вони можуть бути визначені лише під час виконання. Різницю між статичними семантичними проблемами та синтаксичними проблемами іноді важко відрізнити.
Код

```
a=b+c;
```

правильний синтаксис у багатьох мовах. Але чи це правильне твердження на C ++?

1. Чи мають значення `b` і `c` значення?
2. Чи було оголошено `b` і `c` як тип, що дозволяє операцію `+`? Або значення `b` і `c` підтримують операцію `+`?
3. Чи сумісне призначення з результатом виразу `b + c`?
4. Чи має заява про призначення належну форму?

Існує безліч запитань, на які потрібно відповісти щодо цього завдання про призначення. На деякі запитання можна відповісти швидше, ніж на інші. Коли програма C ++ компілюється, вона перекладається з C ++ на машинну мову, як описано в попередньому розділі. Питання 2 і 3 - це питання, на які можна відповісти, коли компілюється програма C ++. Однак відповідь на перше запитання може бути невідомою, поки в деяких випадках програма C ++ не виконається. Відповіді на запитання 2 і 3 можна отримати під час компіляції і називаються статичними семантичними проблемами.

Відповідь на запитання 1 є динамічним питанням, і його, можливо, неможливо визначити до часу виконання.
За деяких обставин відповідь на питання 1 може також бути статичним семантичним питанням. Питання 4, безумовно, є синтаксичним питанням.

На відміну від динамічних семантичних проблем, правильний синтаксис програми є статично визначеним. Зазначено інший спосіб, визначення синтаксично допустимої програми може бути виконано без запуску програми. Синтаксис мови програмування задається граматикою. Але перед обговоренням граматики необхідно визначити частини граматики. Термінал або маркер - це символ на мові.

- Термінали C ++, Java та Python: `while,for,(,;,5,b`
- Імена типу `int` та `string`

Ключові слова, типи, оператори, числа, ідентифікатори тощо - це всі лексеми або термінали в мові.
Синтаксична категорія або нетермінал - це набір фраз або рядків лексем, які визначатимуться як символи в мові (термінальні та нетермінальні символи).

- Нетермінали C ++, Java або Python: **<statement>, <expression>, <if-statement>** тощо.
- Синтаксичні категорії визначають такі частини програми, як оператори, вирази, декларації тощо.

Метамова - це мова вищого рівня, що використовується для вказівки, обговорення, опису чи аналізу іншої мови. Англійська мова використовується як метамова для опису мов програмування, але через неоднозначність англійської мови були розроблені більш офіційні метамови. Наступний розділ описує формальну метамову для опису синтаксису мови програмування.

Форма Бекуса-Наура (БНФ)

Backus Naur Format (тобто BNF) є формальною метамовою для опису синтаксису мови. Слово формальне використовується для позначення того, що БНФ є однозначним. На відміну від англійської, мова BNF не є відкритою для наших власних інтерпретацій. Існує лише один спосіб прочитати опис BNF.

Джон Бакус використовував BNF для опису синтаксису Алгола в 1963 році. У 1960 році Джон Бакус і Пітер Наур, автор комп'ютерних журналів, щойно брали участь у конференції, присвяченій Алголу. Повернувшись із подорожі, стало очевидним, що вони мали дуже різні погляди на те, як би виглядав Алгол. В результаті цієї дискусії Джон Бакус працював над методом опису граматики мови. Пітер Наур дещо змінив його. Позначення називається BNF, або форма Backus Naur або іноді нормальна форма Backus.

БНФ складається з набору правил, які мають таку форму:
<syntactic category> ::= a string of terminals and nonterminals

Символ **::=** може бути прочитаний у складі і означає, що синтаксична категорія - це сукупність усіх елементів, що відповідають правій частині правила.
Кілька правил, що визначають одну і ту ж синтаксичну категорію, можуть бути скорочені за допомогою **|** символ, який можна прочитати як **"або"** і означає набір об'єднань.
Це вся мова. Це не дуже великий метамова, але він потужний.

Приклади БНФ

Ось кілька прикладів BNF з Java.

```
< primitive - type > ::= b o o l e a n
< primitive - type > ::= char
```

Синтаксис у вигляді BNF часто скорочується, коли існує множина подібних правил, подібних цим правилам примітивних типів. Скорочено чи ні, значення одне і те ж.

< primitive - type > ::= b o o l e a n | char | byte | short
| int | long | float | ...
< argument - list > ::= < expression > | < argument - list > , < expression >
< selection - statement > ::=
if (< expression >) < statement > |
if (< expression >) < statement > else < statement > |
s w i t c h (< e x p r e s s i o n >) < block >
< method - d e c l a r a t i o n > ::=
< modifiers > < type - specifier > < method declarator >
< throws - clause > < method - body > |
< modifiers > < type - specifier > < method - declarator >
< method - body > |
< type - specifier >< method - declarator >< throws - clause > <method - body > |

< type - specifier >< method - declarator >< method - body >

Цей опис можна описати звичайною мовою:
“Набір оголошень методів є об’єднанням наборів оголошень методів, які явно створюють виняток із тими, які явно не створюють виняток із модифікаторами або без них, приєднаними до їх визначень.”
БНФ набагато легше зрозуміти і він не є однозначним, як цей опис звичайною мовою.

Розширений БНФ (РБНФ)

Оскільки опис синтаксису мови програмування BNF сильно спирається на рекурсію для надання списків елементів, багато визначень використовують такі розширення:

1. `item?` or `[item]` - означає, що елемент є необов'язковим.
2. `item*` or `{item}` - означає, що допустимі нульові або більше випадків появи елемента.
3. `item+` - означає одне або кілька входжень елемента допустимі.
4. Для групування можуть використовуватися дужки.

Контекстно-вільні граматики

БНФ - це спосіб опису граматики мови. Найцікавіші граматики без контексту, тобто зміст будь-якої синтаксичної категорії в реченні не залежить від контексту, в якому воно використовується.

Безконтекстна граматика визначається як чотирарні кортежі:

$$G = (N, T, P, S)$$

де

N - набір символів, які називаються нетерміналами або синтаксичними категоріями.

T - набір символів, які називаються терміналами або жетонами.

P - сукупність підстановок вигляду $n \to \alpha$, де $n \in N$ і $\alpha \in (N \cup T)^*$.

$S \in N$ - це спеціальний нетермінал, який називається початковим символом граматики.

Неофіційно безконтекстна граматика - це набір нетерміналів та терміналів. Для кожного нетерміналу існує одне або більше виробництв із рядками нуля або більше нетерміналів та терміналів праворуч, як описано в описі BNF. Існує один спеціальний нетермінал, який називається початковим символом граматики.

Граматика інфіксних виразів

Безконтекстну граматику для інфіксних виразів можна вказати як $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$

де

$\mathcal{N} = \{E, T, F\}$

$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$

$\mathcal{P}$ is defined by the set of productions

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid identifier \mid number$

Виведення

Речення граматики - це рядок лексем із граматики. Речення належить до мови граматики, якщо воно може бути похідним від граматики. Цей процес називається побудовою деривації. Деривація - це послідовність речень у формі, що починається символом старту граматики і закінчується реченням, яке ви намагаєтесь вивести. Форма речення - це рядок терміналів та нетерміналів із граматики. На кожному кроці у виведенні один нетермінал сентенціальної форми, який називається A, замінюється рядком терміналів та нетерміналів, β, де $A \rightarrow \beta$ - це утворення в граматиці. Для граматики G мова G - це

сукупність речень, які можуть бути похідними від G і зазвичай записуються як L (G).

Тут ми доводимо, що вираз `(5 * x) + y` є членом мови, що визначається граматикою, поданою на *слайді 21* шляхом побудови висновку для нього. Виведення починається з символу старту граматики і закінчується реченням.

$$\underline{E} \Rightarrow \underline{E} + T \Rightarrow \underline{T} + T \Rightarrow \underline{F} + T \Rightarrow (\underline{E}) + T \Rightarrow (\underline{T}) + T \Rightarrow (\underline{T} * F) + T$$
$$\Rightarrow (\underline{F} * F) + T \Rightarrow (5 * \underline{F}) + T \Rightarrow (5 * x) + \underline{T} \Rightarrow (5 * x) + \underline{F} \Rightarrow (5 * x) + y$$

Кожен крок є сентенційною формою. Підкреслений нетермінал у кожній формі сентенції замінюється правою частиною продукту для цього нетерміналу. Виведення відбувається від початкового символу **`E`** до речення **`(5 * x) + y`**. Це доводить, що **`(5 * x) + y`** знаходиться в мові **`L(G)`**, граматики **`G,`** визначеної на *слайді 22*.

ТИПИ ВИВЕДЕННЯ

Речення граматики є дійсним, якщо для нього існує хоча б одне виведене з використанням граматики. Як правило, існує багато різних виведень для певного граматичного речення. Однак є два виведення, які представляють певний інтерес для нас у розумінні мов програмування.

- Саме ліве виведення - Завжди замінюйте крайній лівий нетермінал, коли переходите від однієї форми речення до наступної у виведенні.
- Найправіше виведення - Завжди замінюйте самий правий нетермінал, коли переходите від однієї форми речення до наступної у виведенні.

Виведення речення `(5 * x) + y` на слайді 24 - це крайній лівий висновок. Самий правий висновок для того самого речення:

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + y \Rightarrow T + y \Rightarrow F + y \Rightarrow (E) + y \Rightarrow (T) + y$$
$$\Rightarrow (T * F) + y \Rightarrow (T * x) + y \Rightarrow (F * x) + y \Rightarrow (5 * x) + y$$

ПРЕФІКСНІ ВИРАЗИ

Інфіксні вирази - це вирази, де оператор відображається між операндами. Інший тип виразу називається префіксним виразом. У префіксних виразах оператор постає перед операндами. Інфіксний вираз **`4 + (a - b) * x`** буде записаний **`+4 * -abx`** як префіксний вираз. Префіксні вирази є в якомусь сенсі простіше, ніж інфіксні вирази, оскільки нам не потрібно турбуватися про перевагу операторів. Пріоритет оператора визначається порядком операцій у виразі. Через це дужки в префіксних виразах не потрібні.

ПРЕФІКСНІ ГРАМАТИКИ

Безконтекстну граматику для префіксних виразів можна вказати як $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{E})$ де

$\mathcal{N} = \{E\}$

$\mathcal{T} = \{identifier, number, +, -, *, /\}$

$\mathcal{P}$ is defined by the set of productions

$E \rightarrow + E E \mid - E E \mid * E E \mid / E E \mid identifier \mid number$

Дерева розбору

Граматика G може бути використана для побудови дерева, що представляє речення L (G), мови граматики G. Цей тип дерева називається синтаксичним деревом. Дерево синтаксичного аналізу - це ще один спосіб представлення речення даної мови. Дерево синтаксичного аналізу будується із символом старту граматики біля кореня дерева. Діти кожного вузла в дереві повинні з'являтися праворуч від продукту, а батьківський - ліворуч від того самого виробництва. Програма є синтаксично допустимою, якщо для неї існує дерево синтаксичного аналізу, використовуючи задану граматику.

Хоча в мові, як правило, багато різних похідних від речення, існує лише одне дерево розбору. Це вірно, якщо граматика не є двозначною. Насправді це визначення двозначності в граматиці. Граматика неоднозначна тоді і лише тоді, коли в мові граматики є речення, яке містить більше одного дерева синтаксичного аналізу.

Дерево синтаксичного аналізу речення, виведеного на слайді 22 зображено на рис. 2.1. Зверніть увагу на схожість між деривацією (процесом виведення) та деревом синтаксичного аналізу.

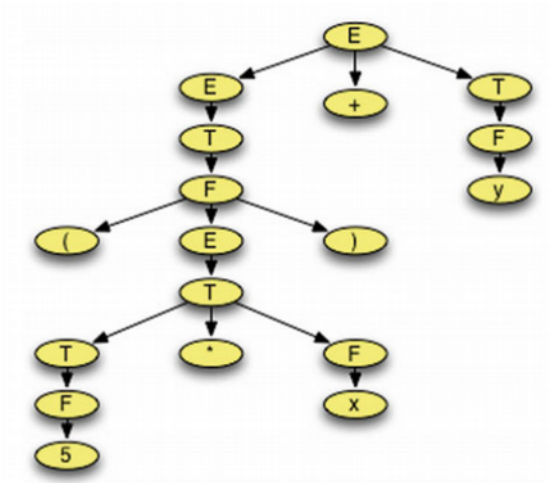

Рис.2.1. Дерево синтаксичного аналізу

АБСТРАКТНІ СИНТАКСИЧНІ ДЕРЕВА

У дереві синтаксичного аналізу є багато інформації, яка насправді не потрібна для фіксації значення програми, яку воно представляє. Абстрактне дерево синтаксису схоже на дерево синтаксичного аналізу, за винятком того, що видаляється несуттєва інформація. Більш конкретно,

Нетермінальні вузли в дереві замінюються вузлами, що зображують частину речення, яку вони представляють.

Елементи проміжного виведення в дереві згортаються.

Наприклад, дерево синтаксичного аналізу з рис. 2.1 можна представити деревом абстрактного синтаксису на рис. 2.2. Абстрактне дерево синтаксису усуває всю непотрібну інформацію і залишає саме те, що є важливим для оцінки виразу. Дерева абстрактних синтаксисів, часто скорочені AST, використовуються компіляторами під час генерації коду і можуть використовуватися інтерпретаторами під час запуску вашої програми. З дерева абстрактних синтаксисів викидають зайву інформацію і зберігають лише те, що є важливим, щоб дозволити компілятору генерувати код або інтерпретатору для виконання програми.

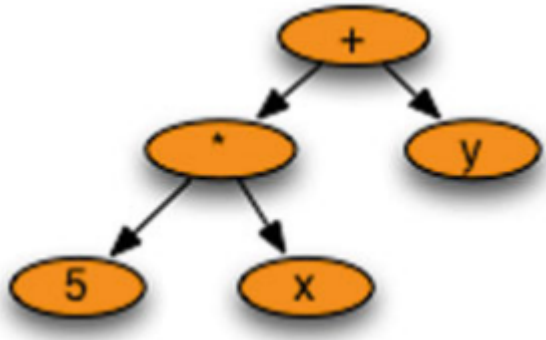

Рис. 2.2. Абстрактне дерево синтаксису.

На наступній лекції ми розглянемо наступні кроки парсінгу: лексичний аналіз, скінченні автомати, безпосередньо алгоритми парсінгу.