ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 3

Лексичний аналіз

Весна 2021

Синтаксис сучасних мов програмування визначається за допомогою граматики. Граматика, оскільки це чітко визначена математична структура, може бути використана для побудови програми, яка називається **парсером**. Реалізація мови, у вигляді компілятора або інтерпретатора, має парсер, який читає програму з вихідного файлу. Синтаксичний аналізатор читає маркери або термінали програми та використовує граматику мови, щоб перевірити, чи не створює потік лексем синтаксично допустиму програму.

Щоб синтаксичний аналізатор виконував свою роботу, він повинен мати можливість отримувати потік токенів з вихідного файлу. Формування токенів з окремих символів вихідного файлу - робота іншої програми, яку часто називають маркером, сканером чи лексером. Лекс - це латинське слово за словом. Слова програми - це її лексеми. У реалізаціях мови програмування трохи свободи береться з визначенням слова. Слово - це будь-який термінал або лексема мови. Виявляється, лексеми мови можна описати іншою мовою, яка називається мовою регулярних виразів.

Мова регулярних виразів

Мова регулярних виразів визначається безконтекстною граматикою. Безконтекстною граматикою для регулярних виразів є R E = ($\mathcal{N}$, $\mathcal{T}$, $\mathcal{P}$, $\mathcal{E}$) де

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

$\mathcal{T}$= {character, ∗, +, ., (,)}

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

E → E + T | T

T → T.K | K

K → F∗ | F

F → character | (E)

Оператор **+** - це оператор вибору, тобто **E** або **T**, але не обидва. Точковий оператор означає, що за **T** слідує **K**. Оператор *****, який називається Kleene Star для математика, який його вперше визначив, означає нуль або більше випадків з **F**. Граматика визначає перевагу цих операторів. Зірка Кліне має найвищий пріоритет, за яким слідує оператор крапок, а потім оператор вибору. На самому примітивному рівні регулярний вираз може бути лише одним символом.

Часто вибір між багатьма різними символами може бути скорочений з якоюсь розумною назвою. Наприклад, буква може використовуватися для скорочення **A + B + · · · + Z + a + b + · · · z**, а цифра може скорочувати **0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9** Зазвичай ці скорочення чітко вказуються перед введенням регулярного виразу.

Лексемами інфіксної граматики є ідентифікатор, число, **+, -, *, /, (,** та**)**. Для вподобань припустимо, що буква та цифра мають звичайні означення. Ми також розмістимо кожен символ оператора в одинарних лапках, щоб не переплутати їх з метамовою. Тоді ці маркери можуть визначатися регулярним виразом

```
letter.letter * + digit.digit * + ‘+’ + ‘-‘ +
‘*’ + ‘/’ + ‘(‘ + ‘)’
```

З цієї специфікації регулярного виразу отримуємо кілька речей. Ідентифікатори повинні мати принаймні один символ, але можуть бути стільки, скільки ми їх бажаємо. Числа - це лише невід'ємні цілі числа в мові вираження інфіксу. Числа з плаваючою комою неможливо вказати мовою, оскільки маркери визначені на даний момент.

Кінцеві автомати

Кінцевий автомат - це математична модель, яка приймає або відхиляє рядки символів для деяких регулярних виразів. Кінцевий автомат часто називають автоматом скінченного стану. Слово автомат - це просто ще одне слово для машини. Кожен регулярний вираз має принаймні один кінцевий автомат і навпаки, кожен кінцевий автомат має щонайменше один відповідний регулярний вираз. Насправді існує алгоритм, за яким будь-який регулярний вираз може бути використаний для побудови кінцевого автомата для нього.

Формально кінцеві автомати визначаються наступним чином. **M = (Σ , S, F, s_0, δ)** де Σ (вимовляється сигма) - вхідний алфавіт (символи, зрозумілі машиною), **S** - набір станів, **F** - підмножина **S**, яка зазвичай записується як **F ⊆ S**, s_0 - це спеціальний стан, який називається стартовим станом, а **δ** (вимовляється дельта) - це функція, яка приймає в якості введення символ алфавіту та стан і повертає новий стан. Зазвичай це записується як **δ: Σ × S → S**.

Кінцевий автомат має поточний стан, який спочатку є стартовим станом. Машина запускається в стартовому стані і читає символи по черзі. Під час зчитування символів кінцевий автомат змінює стан. Кожен стан має перехід до інших станів на основі останнього прочитаного символу. Кожного разу, коли машина переходить у новий стан, з потоку символів зчитується інший символ.

Після прочитання всіх символів маркера, якщо поточний стан знаходиться в наборі кінцевих станів, **F**, то маркер приймається кінцевим автоматом. В іншому випадку він відхиляється. Кінцеві автомати, як правило, представляються графічно, малюючи стани, переходи, стартовий стан і кінцеві стани. Стани в графічному поданні зображуються як вузли на графіку. У стартовому стані в нього входить стрілка, на зворотному боці якої немає нічого. Переходи представлені у вигляді стрілок, що переходять з одного стану в інший, і позначені символами, що ініціюють даний перехід. Нарешті, остаточний або прийнятний стани позначаються подвійним колом.

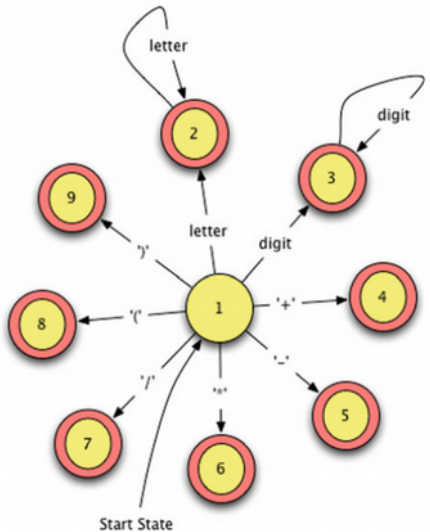


Рис. 3.1 Кінцевий автомат

Малюнок 3.1 зображує кінцевий автомат для мови лексем інфіксних виразів. Початковий стан дорівнює 1. Кожен із станів від 2 до 9 приймає стани, позначені подвійним колом. Держава 2 приймає маркери ідентифікаторів. Держава 3 приймає числові жетони. Штати з 4 по 9 приймають оператори та маркери в дужках. Кінцевий автомат приймає по одному маркеру за раз. Для кожного нового маркера кінцевий автомат запускається спочатку в стані 1.

Якщо під час читання маркера зчитується несподіваний символ, то потік маркерів кінцевий автомат відхиляє як недійсний. В якості лексем приймаються лише дійсні рядки символів. Такі символи, як пробіли, вкладки та символи нового рядка, не розпізнаються кінцевим автоматом. Кінцевий автомат відповідає лише так, рядок лексем відповідає мові, прийнятій машиною, чи ні, це не так.

Лексичні генератори

Порівняно легко побудувати лексер, написавши регулярний вираз, намалювавши кінцевий автомат, а потім написавши програму, що імітує кінцевий автомат. Однак цей процес здебільшого однаковий для всіх мов програмування, тому для цього є інструменти, написані для цього. Зазвичай ці інструменти називаються **генераторами лексера**. Щоб використовувати генератор lexer, ви повинні написати регулярні вирази для лексем мови та надати їх генератору lexer.

Генератор лексера генерує програму лексера, яка внутрішньо використовує кінцевий автомат, такий як той, що зображений на рис. 3.1, але замість повідомлення так чи ні, для кожного маркера лексер повертає рядок символів, що називається лексемою або словом маркер разом із класифікацією лексеми. Отже, ідентифікатори класифікуються як маркери ідентифікаторів, тоді як ‘+’ класифікується як маркер додавання.

Інструмент lex є прикладом лексичного генератора для мови C. Якщо ви пишете інтерпретатор або компілятор, використовуючи C як мову реалізації, тоді ви використовуєте lex або подібний інструмент для створення вашого лексера. lex був інструментом, що входить до складу оригінальної операційної системи Unix. Альтернатива Linux називається flex. Java, Python, Standard ML та більшість мов програмування мають еквівалентні доступні генератори lexer.

Парсінг

Синтаксичний розбір - це процес виявлення, чи є даний рядок лексем дійсним реченням граматики. Кожного разу, коли ви компілюєте програму або запускаєте програму в інтерпретаторі, програма спочатку аналізується за допомогою парсера. Коли синтаксичний аналізатор не може проаналізувати програму, програмісту повідомляють, що в програмі є синтаксична помилка.

Синтаксичний аналізатор - це програма, яка подає речення, перевіряє, чи є це речення мовою даної граматики. Синтаксичний аналізатор зазвичай не просто відповідає так чи ні. Синтаксичний аналізатор часто створює абстрактне представлення дерева синтаксису вихідної програми. Існує два типи синтаксичних аналізаторів, які зазвичай створюються.

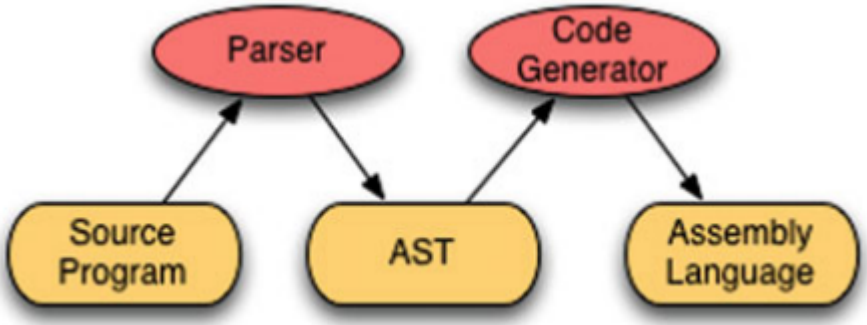


Рис.3.2 Парсер потоку даних

- Синтаксичний розбір зверху вниз починається з кореня дерева синтаксичного аналізу.
- Аналізатор знизу вгору починається з листя дерева синтаксичного аналізу.

Синтаксичні аналізатори зверху вниз та знизу вгору перевіряють, чи належить речення до граматики, будуючи деривацію для речення, використовуючи граматику. Синтаксичний аналізатор або повідомляє про успіх (і, можливо, повертає абстрактне дерево синтаксису), або повідомляє про помилку (сподіваємося, з гарним повідомленням про помилку). Потік даних зображено на рис. 3.2.

Парсери “зверху-вниз”

Аналізатори зверху вниз, як правило, пишуться від руки. Їх іноді називають рекурсивними синтаксичними аналізаторами, оскільки їх можна записати як набір взаємно рекурсивних функцій. Синтаксичний аналізатор зверху вниз виконує крайне ліве виведення речення (тобто вихідну програму).

Аналізатор зверху вниз працює (можливо) переглядаючи наступний маркер у вихідному файлі та вирішуючи, що робити на основі маркера та де він знаходиться у деривації. Для коректної роботи парсер зверху вниз повинен бути розроблений із використанням спеціального виду граматики, що називається граматикою LL (1). Граматика LL (1) - це просто граматика, де наступний вибір у крайньому лівому виведенні може бути детерміновано обраний на основі поточної форми сентенції та наступного маркера у вхідних даних.

Перший L стосується сканування вводу зліва направо. Другий L означає, що, виконуючи крайню ліву деривацію, існує лише 1 символ lookahead, який необхідний для прийняття рішення про те, яке виробництво обрати наступним у деривації.

Граматика LL (1)

Граматикою для префіксних виразів є LL (1). Вивчіть префікс граматики виразу $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{E})$ де

$\mathcal{N}$= {E}

$\mathcal{T}$= {identi f ier, number, +, −, ∗, /}

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

E → + E E | − E E | ∗ E E | / E E | identifier | number

Під час побудови будь-якого виведення для речення цієї мови, наступна продукція, вибрана в самому лівому виведенні, буде очевидною, оскільки наступний маркер вихідного файлу повинен відповідати першому терміналу у вибраній продукції.

Не LL (1) граматика

Деякі граматики не є LL (1). Граматика для інфіксних виразів не є LL (1).

Переглянемо інфіксований вираз граматики $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{E})$ де

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

$\mathcal{T}$= {identi f ier, number, +, −, ∗, /, (,)}

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

E → E + T | E − T | T

T → T ∗ F | T / F | F

F → (E) | identifier | number

Розглянемо інфіксний вираз **5 * 4**. Крайшим лівим виведенням цього виразу буде

E ⇒ T ⇒ T * F ⇒ F * F ⇒ 5 * F ⇒ 5 * 4

Подумайте, дивлячись лише на **5** у виразі. Ми повинні вибрати, чи використовувати виробничий **E → E + T** або **E → T**. Нам дозволено дивитись лише на **5** (тобто ми не можемо дивитися далі **5**, щоб побачити оператор множення). Яке виробництво ми обираємо? Ми не можемо прийняти рішення на основі **5**. Тому граматика не є LL (1).

Те, що ця інфіксна граматика виразів не є LL (1), не означає, що інфіксні вирази не можуть бути проаналізовані за допомогою синтаксичного аналізатора зверху вниз. Існують інші граматики виразів з інфіксом, які є LL (1). Загалом, можна перетворити будь-яку безконтекстну граматику на граматику LL (1). Це можливо, але отримана граматика не завжди легко зрозуміла.

Інфіксна граматика, подана в Розділі. 2.9.2 залишається рекурсивним. Тобто він містить виробництво E → E + T та інше подібне виробництво термінів у виразах з інфіксом. Ці правила залишаються рекурсивними. Ліві рекурсивні правила заборонені в граматиках LL (1). Ліве рекурсивне правило може бути усунене в граматиці шляхом прямолінійної трансформації його виробництва.

Поширені префікси в правій частині двох версій для одного і того ж нетерміналу також не допускаються в граматиці LL (1). Інфіксна граматика, подана раніше *на слайді 28* не містить загальних префіксів. Загальні префікси можна усунути, ввівши в граматику новий нетермінал, замінивши всі загальні префікси новим нетерміналом, а потім визначивши один новий випуск, щоб новий нетермінал складався із загального префікса.

LL(1) граматика інфіксних виразів

Наступна граматика - це граматика LL (1) для інфіксних виразів. $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{E})$ де

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

$\mathcal{T}$= {identi f ier, number, +, −, ∗, /, (,)}

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

E → T Rest E

Rest E → + T Rest E | − T Rest E | ϱ

T → F Rest T

Rest T → ∗ F Rest T | / F Rest T | ϱ

F → (E) | identifier | number

У цій граматиці ϱ (вимовляється епсилон) є спеціальним символом, що позначає порожнє виробництво. Порожнє виробництво - це виробництво, яке не споживає жодних жетонів. Порожні постановки часом зручні в рекурсивних правилах.

Як тільки загальні префікси та ліві рекурсивні правила будуть усунені з контекстної граматики, граматика буде LL (1). Однак це перетворення зазвичай не виконується, оскільки існують більш зручні способи побудови синтаксичного аналізатора навіть для граматик, що не мають рівня LL (1).

Парсери “знизу-догори”

Хоча оригінальна мова вираження інфіксів не є LL (1), це LALR (1). Насправді більшість граматик мов програмування - це LALR (1). LA означає вигляд вперед, 1 означає лише один символ погляду вперед. LR відноситься до сканування вхідних даних зліва направо при побудові самого правого виведення. Синтаксичний аналізатор знизу вгору створює праворуч виведення вихідної програми в зворотному порядку. Отже, синтаксичний аналізатор LALR (1) створює зворотний самий правий висновок програми.

Побудова синтаксичного аналізатора знизу вгору - це дещо складне завдання, що включає обчислення наборів елементів, наборів перспектив, кінцевий автомат і стек. Кінцевий автомат і стек разом називаються автоматом зниження. Побудова автомата зниження та набори перспектив обчислюються з граматики. Знизу вгору парсери зазвичай не пишуться від руки. Натомість використовується генератор синтаксичного аналізатора для створення програми синтаксичного аналізу з граматики.

Генератор синтаксичного аналізатора - це програма, яка отримує граматику та створює синтаксичний аналізатор для мови граматики, будуючи набір автоматів, що розкриваються, і пошукові набори, необхідні для синтаксичного аналізу програм мовою граматики.

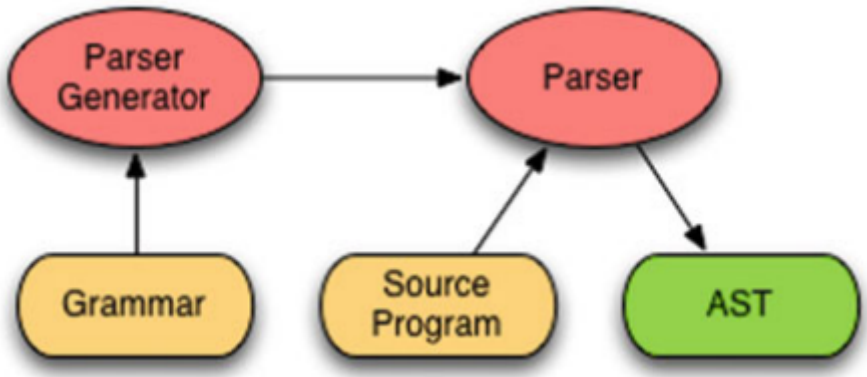


Рис.3.3. Парсер генератора потокових даних

Оригінальний генератор синтаксичного аналізатора для Unix називався **yacc**, що означає ще один компілятор компілятора, оскільки він був компілятором для граматик, який створив парсер для мови. Оскільки парсер є частиною компілятора, **yacc** був компілятором компілятора. Версія **yacc** для Linux називається Bison. Сподіваємось, ви бачите каламбур, який був використаний при іменуванні його Бізон. Генератор синтаксичного аналізатора Bison генерує парсер для компіляторів, реалізованих на C, C ++ або Java. Є версії **yacc** і для інших мов. Standard ML має версію, яку називають **ml-yacc** для компіляторів, реалізовану в Standard ML.

Генератори синтаксичного аналізатора, такі як Бізон, створюють так званий парсер знизу вгору, тому що самий правильний висновок побудований у зворотному порядку. Іншими словами, виведення здійснюється знизу вгору. Зазвичай аналізатор знизу вгору повертає AST, що представляє успішно проаналізовану вихідну програму. На рисунку 3.3 зображений потік даних в інтерпретаторі або компіляторі. Генератору синтаксичного аналізатора дається граматика і він запускається один раз для побудови парсера. Згенерований синтаксичний аналізатор запускається кожного разу, коли аналізується вихідна програма.

Синтаксичний аналізатор “знизу-вгору” аналізує програму шляхом побудови зворотного самого правого виведення вихідного коду. У процесі зворотного виведення синтаксичний аналізатор переміщує токени з вхідного сигналу на стек автомата зниження. Потім у різні моменти часу він зменшується, вирішуючи, виходячи з наборів перспективи, що скорочення необхідно.

Парсінг інфіксних виразів

Розглянемо граматику для інфіксних виразів як $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{E})$ де

N = {E, T, F}

T = {identifier, number, +, −, ∗, /, (,)}

P is defined by the set of productions

(1) E → E + T

(2) E → T

(3) T → T ∗ F

(4) T → F

(5) F → number

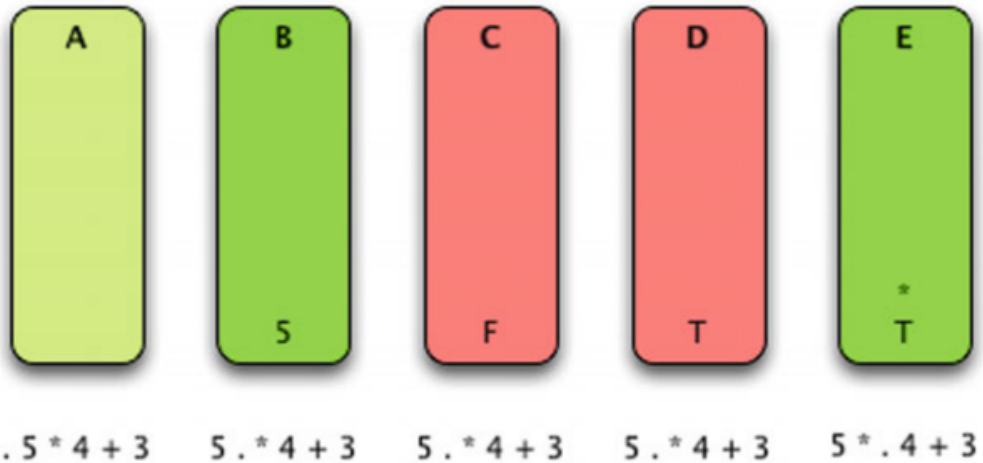
(6) F → (E)

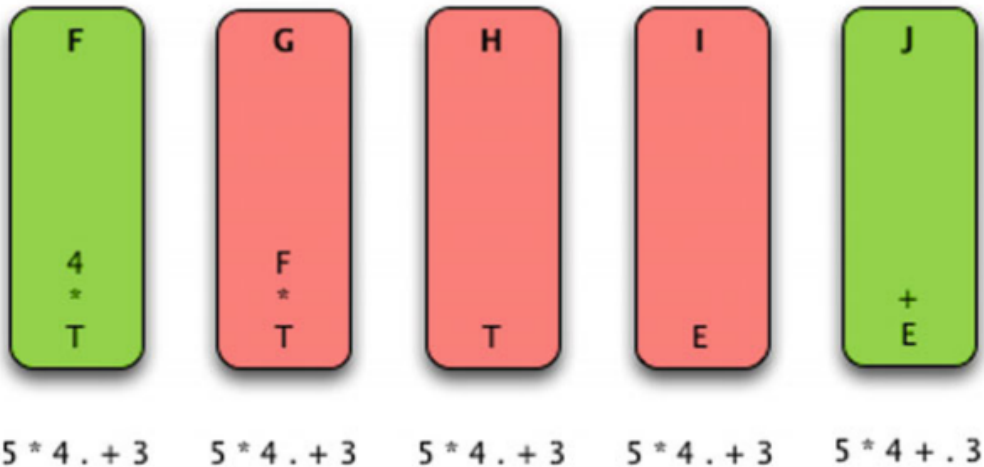
Тепер припустимо, що ми розбираємо вираз $5 * 4 + 3$. Найправіший висновок для цього виразу такий.

$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + 3 \Rightarrow T + 3 \Rightarrow T * F + 3 \Rightarrow T * 4 + 3 \Rightarrow F * 4 + 3 \Rightarrow 5 * 4 + 3$

Синтаксичний аналізатор знизу вгору робить самий правильний вивід у зворотному порядку, використовуючи автомат зниження. Може бути корисним розглянути стек автомата, що розкладається, як він аналізує вираз, як показано на рис. 3.4. На кроці A розпочинається парсер. Крапка ліворуч від 5 означає, що парсер ще не обробив жодних токенів вихідної програми і дивиться на 5. Стек порожній. З кроку A на крок B один маркер, 5 переміщується на стек. З кроку B до C синтаксичний аналізатор дивиться на оператор множення і розуміє, що має бути здійснено скорочення за допомогою правила 5 граматики. Це називається зменшенням, оскільки продукція використовується у зворотному порядку.

Зменшення висуває праву частину правила 5 зі стеку і замінює його нетерміналом F. Якщо ви подивитеся на цей висновок у зворотному порядку, першим кроком є заміна числа 5 на F.


A
B
S
C
F
D
T
E
*
T
.5*4+3
5.*4+3
5.*4+3
5.*4+3
5*.4+3


F
4
*
T
G
F
*
T
H
T
I
E
J
+
E
5 * 4 . + 3
5 * 4 . + 3
5 * 4 . + 3
5 * 4 . + 3
5 * 4 + . 3

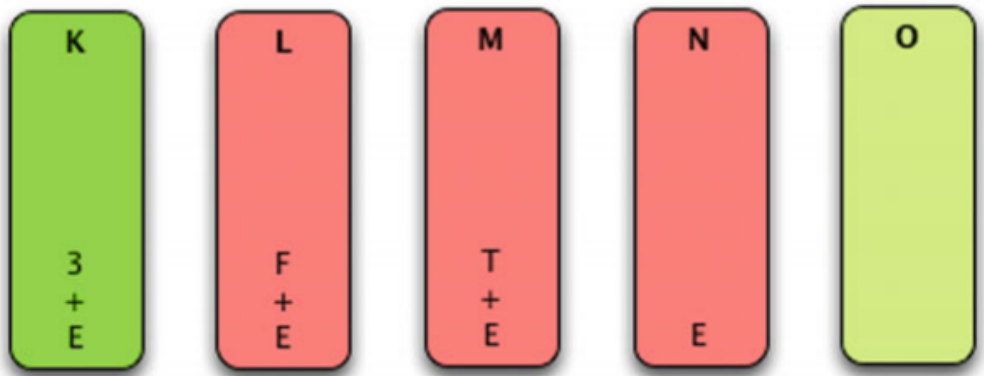

K
3
+
E
L
F
+
E
M
T
+
E
N
E
O
5 * 4 + 3 .
5 * 4 + 3 .
5 * 4 + 3 .
5 * 4 + 3 .
5 * 4 + 3 .

Рис.3.4. Автомат з магазинною пам'яттю стека

Неоднозначність у граматиках

Граматика неоднозначна, якщо існує більше одного дерева синтаксичного аналізу для даного речення мови. Загалом, двозначність у граматиці - це погано. Однак, деякі генератори синтаксичного аналізу для мов LALR (1) можуть допустити певної неясності.

Класичний приклад двозначності в мовах виникає із вкладених тверджень if-then-else. Розглянемо наступне твердження Паскаля:

```
if a<b then
   if b<c then
      writeln("a<c")
else
   writeln("?")
```

Що з твердження відповідає іншому? Це не зовсім зрозуміло. BNF для оператора **`if-then-else`** може виглядати приблизно так.

```
<statement> ::= if <expression> then <statement> else <statement>
              | if <expression> then <statement>
              | writeln ( <expression> )
```

Рекурсивний характер цього правила означає, що оператори **`if-then-else`** можуть бути довільно вкладені. Через це рекурсивне визначення інше в цьому коді бовтається. Тобто незрозуміло, чи відповідає воно першому чи другому твердженню **`if`**.

Коли парсер знизу вгору генерується за допомогою цієї граматики, генератор синтаксичного аналізатора виявить, що граматика має неоднозначність. Проблема проявляється як конфлікт між зміною та операцією скорочення. Перше правило говорить, що при перегляді ключового слова **`else`** синтаксичний аналізатор повинен зміститися. Друге правило говорить, що коли синтаксичний аналізатор дивиться на інше, його слід зменшити.

Для вирішення цього конфлікту, як правило, можна вказати, чи слід створювати синтаксичний аналізатор зсув чи зменшення. Зазвичай за замовчуванням виконується зміщення, і саме це має найбільший сенс у цьому випадку. Переміщаючи, інший варіант буде з найближчим твердженням **`if`**. Це нормальна поведінка синтаксичних аналізаторів, коли стикаються з цією неясністю **`if-then-else`**.

Інші види граматик

Як комп'ютерний програміст, ви, швидше за все, вивчите принаймні одну нову мову і, можливо, кілька під час своєї кар'єри. Нові області застосування часто спричиняють розробку нових мов, щоб зробити програми для програмування в цій області більш зручними. Java, JavaScript та ASP.NET - це три мови, створені завдяки всесвітній мережі.

Ruby та Perl - це мови, які стали популярними мовами розробки для програмування на базі даних та на сервері. Завдання C - ще одна мова, яка стала популярною завдяки зростанню програмування додатків iOS для продуктів Apple. Нещодавня тенденція в мовах програмування полягає у розробці мов для певних доменів для певних вбудованих платформ.

Посібники з мов програмування містять певний довідник, що описує конструкції мови. Багато з цих довідкових посібників дають граматику мови, використовуючи варіацію безконтекстної граматики. Приклади включають граматики CBL (подібні Коболу), синтаксичні діаграми та, як ми вже бачили, BNF та EBNF. Усі ці синтаксичні метамови мають ті самі функції, що й граматики. Всі вони мають певний спосіб визначення частин програми або синтаксичних категорій, і всі вони мають засоби визначення мови за допомогою рекурсивно визначених виробництв. Визначення, поняття та приклади, наведені в цьому розділі, допоможуть вам зрозуміти посилання на мову, коли настане час вивчати нову мову.

Обмеження синтаксичних визначень

Конкретний синтаксис мови майже завжди є неповним описом. Не всі синтаксично допустимі рядки лексем слід розглядати як дійсні програми. Наприклад, розглянемо вираз 5 + 4/0. Синтаксично це допустимий вираз, але, звичайно, не може бути оцінений, оскільки ділення на нуль не визначено. Це семантичне питання. Значення виразу невизначене, оскільки поділ на нуль невизначений. Це семантичне питання, і семантика не описується синтаксичним визначенням. Все, що може забезпечити граматика, це те, що програма є синтаксично допустимою.

Насправді не існує граматики BNF або EBNF, яка генерує лише легальні програми будь-якою мовою програмування, включаючи C ++, Java та Standard ML. Граматика BNF визначає безконтекстну мову: ліва частина кожного правила містить лише одну синтаксичну категорію. Він замінюється одним із альтернативних визначень, незалежних від контексту, в якому це відбувається.

Набір програм будь-якою цікавою мовою не є контекстним. Наприклад, коли обчислюється вираз a + b, чи сумісні типи a та b та визначені над оператором +? Це контекстна проблема, яку неможливо вказати за допомогою контекстної граматики. Контекстно-чутливі функції можна офіційно описати як набір обмежень або контекстних умов. Контекстно-чутливі питання стосуються головним чином декларацій ідентифікаторів та сумісності типів. Іноді такі контекстно-вразливі проблеми, як це, називають частиною статичної семантики мови.

У той час як граматика описує, як лексеми складаються, щоб сформувати дійсну програму, граматика не визначає семантику мови, а також не описує статичну семантику або контекстну характеристику мови. Для опису цих мовних характеристик необхідні інші засоби. Деякі методи, такі як правила виведення типів, формально визначені. Більшість семантичних характеристик неофіційно визначені в якомусь англомовному описі.

Це все контекстно-залежні проблеми.

- У оголошенні масиву на C ++ розмір масиву повинен бути невід'ємним значенням.
- Операнди для операції && повинні бути логічними на Java.
- У визначенні методу повернене значення повинно бути сумісним із типом повернення в декларації методу.
- Коли викликається метод, фактичні параметри повинні відповідати формальним типам параметрів.

Висновки

Цей розділ познайомив вас із синтаксисом мови мов програмування та синтаксичними описами. Читати та розуміти синтаксичні описи варто, оскільки ви, безсумнівно, зіткнетеся з новими мовами у своїй кар'єрі комп'ютерного вченого. Безумовно, можна сказати більше про тему синтаксису мови програмування.

Ахо, Сетхі та Уллман написали широко визнану грунтовну книгу про реалізацію компілятора, яка включає матеріали щодо визначення синтаксису та реалізації синтаксичного аналізатора. Є також багато інших хороших посилань на компілятор. Ієрархія Хомських мов також тісно пов'язана з граматиками та регулярними висловами. Багато книг про дискретні структури в інформатиці вводять цю тему, і кілька хороших книг глибше досліджують ієрархію Хомських, включаючи чудовий текст Пітера Лінца.

У наступних лекціях використовуючи ці знання щодо визначення синтаксису з користю для вивчення нової мови: мови збірки JCoCo. JCoCo - це віртуальна машина для інтерпретації інструкцій байт-коду Python. Вивчення асемблерської мови допомагає краще зрозуміти, як працюють мови вищого рівня та наступні лекції нададуть багато прикладів програм Python та відповідних програм асемблера JCoCo, щоб показати, як реалізована мова високого рівня.

На наступній лекції ми почнемо розглядати мову JCoCo.