ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 5

Мова Ассеблера (продовження)

Весна 2021

If-Then-Else твердження

Мови програмування повинні мати можливість виконувати код на основі умов, що надаються зовні за допомогою введення або обчислюються з інших значень під час виконання програми. Оператори **`if-then`** є одним із засобів умовного виконання коду. Наведений тут код ізолює лише оператор **`if-then`**, щоб показати, як він реалізований у збірці JCoCo.

```
1  import disassembler
2  def main():
3      x=5
4      y=6
5      if x > y:
6          z=x
7      else:
8          z=y
9      print(z)
10 disassembler.disassemble(main)
```

Розбирання цього коду Python призводить до появи коду на рис. 5.1. На Рис. 5.1 є нові вказівки, які досі не зустрічались, але не менш важливо, що в цьому коді є мітки. Етикетка надає символічну ціль, до якої можна перейти в коді. Мітки, як **`label00`** та **`label01`**, визначаються, записуючи їх перед інструкцією, і закінчуються двокрапкою. Мітка праворуч від інструкції є ціллю для цієї інструкції. Етикетки є зручністю для всіх мов складання. Вони дозволяють програмістам асемблерної мови думати скочити до цілі в програмі, а не змінювати вміст реєстру ПК, що відбувається насправді.

Коли програма виконується за допомогою JCoCo, мітки зникають, оскільки JCoCo збирає код, замінюючи мітки фактичними цільовими адресами ПК. Код JCoCo на рис. 5.2 показує код JCoCo після його складання. Зібраний код друкується *кокосом* під час виконання програми.

Рис.5.1.
If-Then-Else
асемблювання

```
1  Function: main/0
2  Constants: None, 5, 6
3  Locals: x, y, z
4  Globals: print
5  BEGIN
6              LOAD_CONST               1
7              STORE_FAST               0
8              LOAD_CONST               2
9              STORE_FAST               1
10             LOAD_FAST                0
11             LOAD_FAST                1
12             COMPARE_OP               4
13             POP_JUMP_IF_FALSE label00
14             LOAD_FAST                0
15             STORE_FAST               2
16             JUMP_FORWARD label01
17 label00:    LOAD_FAST                1
18             STORE_FAST               2
19 label01:    LOAD_GLOBAL              0
20             LOAD_FAST                2
21             CALL_FUNCTION            1
22             POP_TOP
23             LOAD_CONST               0
24             RETURN_VALUE
25 END
```

5.2 Зібраний код

```
Function: main/0
Constants: None, 5, 6
Locals: x, y, z
Globals: print
BEGIN
            LOAD_CONST              1
            STORE_FAST              0
            LOAD_CONST              2
            STORE_FAST              1
            LOAD_FAST               0
            LOAD_FAST               1
            COMPARE_OP              4
            POP_JUMP_IF_FALSE       11
            LOAD_FAST               0
            STORE_FAST              2
            JUMP_FORWARD            13
            LOAD_FAST               1
            STORE_FAST              2
            LOAD_GLOBAL             0
            LOAD_FAST               2
            CALL_FUNCTION           1
            POP_TOP
            LOAD_CONST              0
            RETURN_VALUE
END
```

Перша інструкція, **LOAD_CONST**, має зсув **0** у коді. Інструкції кожної функції мають нульові зсуви з початку функції, тому ми можемо думати про кожну функцію як про власний адресний простір, що починається з нуля. У коді на рис. 5.1 і 5.2 номер рядка першої інструкції дорівнює **6**, тому **6** можна відняти від номерів рядків, щоб визначити адресу будь-якої інструкції в межах функції, а **6** можна додати до будь-якої цілі, щоб визначити номер рядка цільового розташування.

На рис. 5.2 ціллю рядка 13 є 11, що відповідає рядку 17. Дивлячись на рис. 5.1, це відповідає рядку, де визначено **`label00`**. Аналогічно, ціллю інструкції **`JUMP_FORWARD`** на рис. 5.2 є **`label01`**, яка визначена в рядку 19. Віднімаючи **`6`**, ми очікуємо побачити **`13`** як цільову адресу ПК у зібраному коді з рис. 5.2.

Звернувшись до JCoCo BNF у *Додатку А (див.сайт)*, в одній інструкції може бути кілька ярликів. Крім того, адреси інструкцій не мають нічого спільного з тим, в якому рядку вони знаходяться. Це виглядає лише на рис. 5.2, оскільки інструкції розташовані в послідовних рядках. Але додавання порожніх рядків до програми нічого не змінить для зміни адрес інструкцій. Отже, ми могли б мати таку програму, де одна інструкція має дві мітки. Ці три інструкції будуть розміщені за трьома адресами в програмі, хоча в коді є чотири рядки.

```
  onelabel:  LOAD_FAST      1
             STORE_FAST     2
  twolabel:
threelabel:  LOAD_GLOBAL    0
```

Етикетки можуть складатися з будь-якої послідовності букв, цифр, підкреслення або символу `@`, але повинні починатися з літери, підкреслення або символу `@`. Вони можуть мати будь-яку кількість символів.

На рис. 5.1, рядки 6–11 завантажують два значення, що підлягають порівнянню, у стеку. Інструкція **COMPARE_OP** у рядку **12** має аргумент **4**. Звернення до інструкції **COMPARE_OP** у Додатку А показує, що **4** відповідає більшому, ніж порівняння. Порівняння здійснюється за допомогою виклику магічного методу **__gt__** для другого елемента зверху стека операндів і передачі його вершині стеку операндів. Два операнди витиснюються інструкцією **COMPARE_OP**, і в результаті булеве значення **True** або **False** виводиться на стек операндів.

Наступна інструкція переходить до цільового розташування, якщо значення, що залишилось у стеку операндів, було **`False`**. У будь-якому випадку, інструкція **`POP_JUMP_IF_FALSE`** вискакує верхнє значення зі стеку операндів.

Зверніть увагу на рядок 16 на рис. 5.1. У збірці немає нічого подібного до оператора **`if-then-else`**. Рядок 15 - кінець коду, який реалізує тодішню частину виписки. Без рядка 16 JCoCo продовжував би виконувати і перейшов би до іншої частини заяви. Інструкція **`JUMP_FORWARD`** необхідна для переходу через іншу частину коду, якщо тодішня частина була виконана. Рядок 17 починає код **`else`**, а рядок 18 є останньою інструкцією оператора **`if-then-else`**. Визначення мітки для **`label01`** все ще є частиною оператора **`if-then-else`**, але позначає інструкцію безпосередньо після оператора **`if-then-else`**.

If-Then твердження

Часто твердження **`if-then`** пишуться без пропозиції **`else`**. Наприклад, ця програма друкує **`x`**, якщо **`x`** більше, ніж **`y`**. В обох випадках друкується **`y`**.

```
1  import disassembler
2
3  def main():
4      x = 5
5      y = 6
6      if x > y:
7          print(x)
8
9      print(y)
10
11 disassembler.disassemble(main)
```

Розбирання цього коду створює програму на рис. 5.3. Код дуже схожий на код, представлений на рис. 5.1. Рядок 13 ще раз проскакує повз тодішню частину програми. Рядки 14–17 містять тодішній код. Цікаво, що рядок 18 переходить вперед до рядка 19. Порівнюючи це з кодом на рис. 5.1, де перехід вперед стрибає повз іншу частину, те саме відбувається на рис. 5.3, за винятком того, що іншої частини висловлювання немає.

Деякі мови збірки не мають еквівалента **`POP_JUMP_IF_FALSE`**. Натомість доступний лише еквівалент **`POP_JUMP_IF_TRUE`**. У цьому випадку можна перевірити протилежність умови, і стрибок буде виконаний, якщо вірно протилежне, пропускаючи тодішню частину. Наприклад, якщо тестування на більше, ніж є наміром коду, може бути протестовано менше або рівне для переходу до тодішньої частини оператора **`if-then-else`**.

Рис.5.3 If-Then збірка

```
Function: main/0
Constants: None, 5, 6
Locals: x, y
Globals: print
BEGIN
            LOAD_CONST              1
            STORE_FAST              0
            LOAD_CONST              2
            STORE_FAST              1
            LOAD_FAST               0
            LOAD_FAST               1
            COMPARE_OP              4
            POP_JUMP_IF_FALSE label00
            LOAD_GLOBAL             0
            LOAD_FAST               0
            CALL_FUNCTION           1
            POP_TOP
            JUMP_FORWARD      label00
label00:    LOAD_GLOBAL             0
            LOAD_FAST               1
            CALL_FUNCTION           1
            POP_TOP
            LOAD_CONST              0
            RETURN_VALUE
END
```

Незалежно від того, тестували початковий стан чи навпаки, очевидно, **JUMP_FORWARD** не потрібен у коді на рис. 5.3. Ми вже бачили, що компілятор Python може сформувати марнотратну інструкцію. Не неправильно стрибати вперед, це просто не потрібно. Зручність написання такою мовою, як Python, набагато перевищує незручності написання такою мовою, як мова асемблера JCoCo, тому додаткові вказівки час від часу це не така вже й велика проблема. У цьому випадку, компілятор Python може бути написаний таким чином, щоб визнати, коли додаткові інструкції не потрібні.

Цикл While

Розглянемо цей код, який обчислює число Фібоначчі для значення, що зберігається у змінній **`f`**. Послідовність чисел Фібоначчі обчислюється додаванням двох попередніх чисел у послідовності, щоб отримати наступне число. Послідовність складається з **`1, 1, 2, 3, 5, 8, 13, 21`** тощо, восьмий елемент послідовності — **`21`**.

```
1  import disassembler
2  def main():
3    f=8
4    i=1
5    j=1
6    n=1
7    while n < f:
8      n=n+1
9      tmp = j
10     j=j+i
11     i = tmp
12   print("Fib("+str(n)+") is",i)
13 disassembler.disassemble(main)
```

Збірка JCoCo для цієї програми реалізує цикл **`while`** програми Python, використовуючи інструкції **`JUMP_ABSOLUTE`** та **`POP_JUMP_IF_FALSE`**. До циклу призначення інструкції **`SETUP_LOOP`** не є очевидним. У Python з циклу можна вийти, використовуючи інструкцію **`break`**. Використання **`break`** у циклі не є рекомендованим стилем програмування. Переривання ніколи не потрібна. Це іноді використовується як зручність. Щоб обробляти інструкцію **`break`**, коли вона виконується, повинні бути певні знання про те, де закінчується цикл.

У коді на рис. 5.4 перша інструкція після циклу знаходиться в рядку 33, де визначено **`label02`**. Інструкція **`SETUP_LOOP`** штовхає адресу цієї інструкції до стеку блоків. Якщо виконується інструкція про розрив, блок стеків висмикується, і ПК встановлюється на висмикувану інструкцію адресу.

```
Function: main/0
Constants: None,8,1,"Fib(",") is"
Locals: f, i, j, n, tmp
Globals: print, str
BEGIN
            LOAD_CONST              1
            STORE_FAST              0
            LOAD_CONST              2
            STORE_FAST              1
            LOAD_CONST              2
            STORE_FAST              2
            LOAD_CONST              2
            STORE_FAST              3
            SETUP_LOOP label02
label00:    LOAD_FAST               3
            LOAD_FAST               0
            COMPARE_OP              0
            POP_JUMP_IF_FALSE label01
            LOAD_FAST               3
            LOAD_CONST              2
            BINARY_ADD
            STORE_FAST              3
            LOAD_FAST               2
```

Рис.5.4
Збірка While
цикла

```
24              STORE_FAST               4
25              LOAD_FAST                2
26              LOAD_FAST                1
27              BINARY_ADD
28              STORE_FAST               2
29              LOAD_FAST                4
30              STORE_FAST               1
31              JUMP_ABSOLUTE label00
32  label01:    POP_BLOCK
33  label02:    LOAD_GLOBAL              0
34              LOAD_CONST               3
35              LOAD_GLOBAL              1
36              LOAD_FAST                3
37              CALL_FUNCTION            1
38              BINARY_ADD
39              LOAD_CONST               4
40              BINARY_ADD
41              LOAD_FAST                1
42              CALL_FUNCTION            2
43              POP_TOP
44              LOAD_CONST               0
45              RETURN_VALUE
46  END
```

Рядки 15–18 на рис. 3.6 реалізують порівняння **`n <f`** подібно до того, як виконуються порівняння **`if-then-else`**. Перший рядок цього коду позначений **`label00`**, оскільки кінець циклу відскакує туди, щоб побачити, чи слід виконати іншу ітерацію. Цикл **`while`** продовжує виконуватися, доки умова не оцінюється як **`False`**, тому інструкція **`POP_JUMP_IF_FALSE`** переходить до **`label01`**, коли цикл завершується.

Інструкція на **label01** позначає інструкцію **POP_BLOCK**. Ця інструкція потрібна, якщо цикл виходить нормально, а не як результат оператора **break**. Стек блоків вискакується, видаляючи з нього точку виходу з циклу. При виході в результаті перерви, виконання переходить до інструкції в рядку 33, пропускаючи інструкцію **POP_BLOCK**, оскільки оператор **break** вже вибив стек блоку.

Важливо зауважити, що цикл **while** та оператор **if-then-else** реалізуються з використанням тих самих інструкцій. Спеціальної інструкції з циклу на мові асемблеру немає. Загальний потік циклу **while** - це тест перед тілом циклу, що відповідає умові циклу **while**. Якщо умова циклу не виконується, виконання переходить до наступної інструкції після циклу. Після тіла циклу перехід повертає виконання до коду умови циклу **while**, щоб перевірити, чи є інший буде виконана ітерація тіла. Ця ідіома, або шаблон інструкцій, використовується для реалізації циклів, а подібні шаблони використовуються і для циклів в інших мовах збірки.

Обробка винятків

Обробка винятків відбувається в Python в операторі **try-osim**. Заяви всередині блоку **try** виконуються, і якщо виникає виняток, виконується перехід до блоку операторів **same**. Якби **main** викликали програму Python, наведену тут, будь-яка умова помилки надсилала б її в блок **exclude**, який просто друкує виняток у цьому випадку. Блок "виключення" виконується лише в тому випадку, якщо в блоці "спроба" є помилка.

Помилки, які можуть виникнути в цій програмі, можуть бути помилкою перетворення для будь-якого з двох перетворень числа з плаваючою точкою або поділом на нуль. Код вловлює виняток, якщо для другого значення введено нуль.

```
1  import disassembler
2  def main():
3      try:
4          x = float(input("Enter a number: "))
5          y = float(input("Enter a number: "))
6          z=x/y
7          print(x,"/",y,"=",z)
8      except Exception as ex:
9          print(ex)
10 disassembler.disassemble(main)
```

Реалізація обробки винятків у JCoCo дещо схожа на реалізацію інструкції **`BREAK_LOOP`**. Різниця полягає в тому, що виняток змушує програму переходити з одного місця на інше замість інструкції **`BREAK_LOOP`**. Як обробка винятків, так і інструкція про розрив використовують стек блоків. Коли вводиться цикл, інструкція **`SETUP_LOOP`** штовхає точку виходу з циклу на стек блоків; точка виходу - ціле число, що посилається на адресу першої інструкції після циклу.

Щоб розрізнити точки виходу з циклу та обробку винятків, інструкція **SETUP_EXCEPT** штовхає мінус адреси за винятком обробника (тобто **–1 * адреса**). Отже, від'ємне число в блоці блоків відноситься до обробника винятків, тоді як позитивне значення стосується точки виходу з циклу. У коді на рис. 5.5

Код обробника винятків починається з **label00**.

Код спроби блоку починається з рядка 7 з **SETUP_EXCEPT**. Це виштовхує адресу обробника для **label00** у стеці блоків, що відповідає **–27**.

Виконання триває шляхом отримання вхідних даних від користувача, перетворення вхідних даних у плаваючі, виконання поділу та друк результату. Друк завершується в рядку 24, де `None`, який повертається друком, вискакується зі стеку операндів.

Якщо виконання доходить до кінця блоку спроб, то не сталося винятків, і рядок 25 виводить `-27` із стеку блоків, закінчуючи блок спроби. Рядок 26 стрибає за кінець блоку.

Якщо виникає виняток, три речі переміщуються в стек операндів до того, як відбувається будь-яка обробка винятку. Трекбек натискається першим. Відстеження - це копія стеку часу виконання, що містить кожен виклик функції та збережений ПК усіх функцій, що очікують, включаючи кадр стеку поточної функції та ПК. Над зворотним відстеженням є дві копії об'єкта виключення, висунутого на стек операндів, коли виникає виняток.

Якщо в блоці try виникає виняток, JCoCo звертається до стеку блоків і вискакує значення, поки не буде знайдена негативна адреса, яка відповідає деякому крім блоку. Можуть бути вкладені кілька операторів спроби за винятком, тому цілий стек блоків міститиме більше однієї негативної адреси. Коли знайдено негативну адресу, для ПК встановлюється позитивне значення, через що виконання переходить до блоку за винятком. На рис. 5.5, це рядок **27**. Відстеження та дві копії винятку виштовхуються в стек до того, як буде виконано рядок **27**.

Чому три об’єкти виштовхуються на стек операндів, коли виникає виняток? Інструкція **`RAISE_VARARGS`** Python описує вміст стеку операндів як **`TOS2`**, що містить зворотну трасування, **`TOS1`** параметр і **`TOS`** виняток. У реалізації JCoCo параметр до винятку можна отримати, перетворивши виняток у рядок, тому об'єкт у **`TOS1`** знову просто виняток. Для сумісності з розбірником Python JCoCo штовхає три операнди до стеку операндів, коли викликається виняток.

Обробники винятків у Python можуть бути написані для відповідності лише певним типам винятків. Наприклад, у Python поділ на нульові винятки відрізняється від помилки перетворення з плаваючою точкою. На сьогодні віртуальна машина JCoCo має лише один тип винятків, який називається **`Exception`**.

Можна розширити JCoCo для підтримки інших типів винятків, але наразі існує лише один тип об’єкта виключення, який можна створити. Аргументом об’єкта виключення може бути все, що завгодно. Програма на рис. 5.5 написана для лову будь-якого типу винятків, але вона може бути написана для лову лише певного типу винятків. Рядок 27 дублює об’єкт виключення у верхній частині стеку операндів. Рядок 35 завантажує глобальний об'єкт **`Exception`** у стек.

Інструкція **`COMPARE_OP 10`** порівнює виняток, використовуючи порівняння збігів винятків, яке викликає магічний метод **`__excmatch__`**, щоб побачити, чи існує збіг між викинутим винятком та вказаним шаблоном. Якщо збігу немає, рядок 30 переходить до кінця блоку, крім. Інструкція **`END_FINALLY`** у рядку 47 визначає, чи було оброблено виняток, а якщо ні, то повторно видає виняток для якогось зовнішнього блоку обробки винятків.

```
Function: main/0
Constants: None,
    "Enter a number: ", "/", "="
Locals: x, y, z, ex
Globals: float,input,print,Exception
BEGIN
          SETUP_EXCEPT label100
          LOAD_GLOBAL          0
          LOAD_GLOBAL          1
          LOAD_CONST           1
          CALL_FUNCTION        1
          CALL_FUNCTION        1
          STORE_FAST           0
          ...
          BINARY_TRUE_DIVIDE
          STORE_FAST           2
          LOAD_GLOBAL          2
          LOAD_FAST            0
          LOAD_CONST           2
          LOAD_FAST            1
          LOAD_CONST           3
          LOAD_FAST            2
          CALL_FUNCTION        5
          POP_TOP
          POP_BLOCK
```

5.5 Збірка обробки винятків

```
26              JUMP_FORWARD label03
27  label00:    DUP_TOP
28              LOAD_GLOBAL           3
29              COMPARE_OP           10
30              POP_JUMP_IF_FALSE label02
31              POP_TOP
32              STORE_FAST            3
33              POP_TOP
34              SETUP_FINALLY label01
35              LOAD_GLOBAL           2
36              LOAD_FAST             3
37              CALL_FUNCTION         1
38              POP_TOP
39              POP_BLOCK
40              POP_EXCEPT
41              LOAD_CONST            0
42  label01:    LOAD_CONST            0
43              STORE_FAST            3
44              DELETE_FAST           3
45              END_FINALLY
46              JUMP_FORWARD label03
47  label02:    END_FINALLY
48  label03:    LOAD_CONST            0
49              RETURN_VALUE
50  END
```

Якщо винятком було збіг, виконання коду обробника розпочинається так само, як і в рядку 31 програми. У верхній частині стеку операндів міститься додатковий об'єкт винятку, тому його викидає рядок 31. Рядок 32 бере решту об'єкта виключення та робить попередню посилальну точку на нього. У рядку 33 з’являється зворотний зв'язок із стеку операндів.
Якщо під час виконання обробника винятків виникає виняток, тоді JCoCo повинен очистити від винятку. Рядок 34 виконує інструкцію **`SETUP_FINALLY`**, щоб проштовхнути інший запис стека блоків, щоб відстежувати кінець обробника винятків. У рядках 35–38 виводиться виняток з іменем **`ex`** у коді.

У рядку 39 з'являється адреса виходу, яка була просунута інструкцією **SETUP_FINALLY**. Інструкція **POP_EXCEPT** у рядку 40 потім вискакує адресу стека блоків для адреси виходу обробника винятків. Рядок 41 висуває **None** у стек операндів.

Рядок 42 - це або наступна виконана інструкція, або вона переходить до неї в результаті винятку під час виконання коду обробника для попереднього винятку. У будь-якому випадку, змінна ex робиться для посилання на **None**. Інструкція **DELETE_FAST**, здається, не робить багато в цьому коді. Він генерується дізасемблером, але, схоже, видаляє **None**, що, здається, не потрібно робити.

Остання інструкція коду обробника, інструкція **`END_FINALLY`** перевіряє, чи було оброблено виняток. У цьому випадку це було оброблено, і інструкція нічого не робить. Якщо виконання переходить до рядка 47, тоді обробник винятків не відповідає піднятому винятковій ситуації, і тому виняток повторно піднімається. Рядок 48 завершується, налаштовуючи повернення **`None`** з основної функції.

Константа списку

Скласти складне значення, як список, не надто складно. Щоб побудувати константу списку за допомогою JCoCo, ви натискаєте елементи списку на стеку операндів у тому порядку, в якому ви хочете, щоб вони відображались у списку. Потім ви викликаєте інструкцію **`BUILD_LIST`**. Аргумент інструкції визначає довжину списку. Цей код створює список і друкує його на екран.

```
1 import disassembler
2 def main():
3     lst = ["hello","world"]
4     print(lst)
5 disassembler.disassemble(main)
```

Рис.5.6 Збірка для побудови списку

```
1  Function: main/0
2  Constants: None, "hello", "world"
3  Locals: lst
4  Globals: print
5  BEGIN
6              LOAD_CONST              1
7              LOAD_CONST              2
8              BUILD_LIST              2
9              STORE_FAST              0
10             LOAD_GLOBAL             0
11             LOAD_FAST               0
12             CALL_FUNCTION           1
13             POP_TOP
14             LOAD_CONST              0
15             RETURN_VALUE
16 END
```

Програма мови асемблера на рис. 5.6 створює список із двома елементами: **[„привіт", „світ"]**. Рядки 6 і 7 штовхають дві струни на стеку операндів. Рядок 8 спливає два операнди зі стеку, будує об'єкт списку і виштовхує отриманий список у стек операндів. Python визначає **_ _str_ _** магічний метод для вбудованого типу значення, який викликається у списку в рядку 12.

Якщо ви запустите цю програму за допомогою інтерпретатора JCoCo, ви помітите, що **[„привіт", „світ"]** не друкується на екрані. Натомість друкується **[привіт, світ]**. Це відбувається тому, що в даний час метод **_ _str_ _** викликається для кожного елемента списку, щоб перетворити його на рядок для друку. Це не правильний спосіб викликати. Натомість слід викликати магічний метод **_ _repr_ _**, який повертає подане для друку подання значення, що зберігає будь-яку інформацію про тип.

Виклик методу

Викликати такі функції, як друк та введення, було відносно просто. Надішліть ім'я функції з аргументами до функції у стеку операндів. Потім викличте функцію за допомогою інструкції **`CALL_FUNCTION`**. Але як щодо методів? Як метод, такий як **`split`**, викликається на рядку? Ось програма, яка демонструє, як викликати **`split`** у Python.

```
1  Function: main/0
2  Constants: None,"Enter integers:"
3  Locals: s, lst
4  Globals: input, split, print
5  BEGIN
6              LOAD_GLOBAL              0
7              LOAD_CONST               1
8              CALL_FUNCTION            1
9              STORE_FAST               0
10             LOAD_FAST                0
11             LOAD_ATTR                1
12             CALL_FUNCTION            0
13             STORE_FAST               1
14             LOAD_GLOBAL              2
15             LOAD_FAST                1
16             CALL_FUNCTION            1
17             POP_TOP
18             LOAD_CONST               0
19             RETURN_VALUE
20 END
```

Рис.3.7 Збірка виклику метода

```
1 import disassembler
2 def main():
3     s = input("Enter integers:")
4     lst = s.split()
5     print(lst)
6 disassembler.disassemble(main)
```

Рядок 6 коду асемблерної мови на рис. 5.7 готується викликати функцію введення, завантажуючи введені імена в стек операндів. Рядок 7 завантажує аргумент у вхідний рядок підказки. Рядок 8 викликає функцію введення, залишаючи введений текст у стеку операндів. Подібний виклик здійснюється аналогічно.

У цьому коді Python синтаксис виклику введення та розбиття зовсім інший. Python бачить різницю і використовує інструкцію **`LOAD_ATTR`** мовою асемблера інструкції для отримання атрибута **`split`** об'єкта, на який посилається **`s`**. Рядок 10 завантажує об’єкт, на який посилається **`s`**, у стек. Потім рядок 11 знаходить атрибут **`split`** цього об’єкта.

Кожен об’єкт у JCoCo та Python містить словник усіх атрибутів об’єкта. Ця інструкція **`LOAD_ATTR`** вивчає словник та ключ, який знаходиться в глобальному списку в індексі операндів. Потім він завантажує цей атрибут у стек операндів. Потім інструкція **`CALL_FUNCTION`** викликає метод, який знаходився з інструкцією **`LOAD_ATTR`**.

Інструкція **STORE_ATTR** зберігає атрибут в об'єкті приблизно так само, як завантажується атрибут. На даний момент JCoCo не підтримує інструкцію **STORE_ATTR**, але може з відносно невеликими зусиллями. Можливість завантажувати та зберігати атрибути об'єкта означає, що JCoCo може бути використаний для реалізації об'єктно-орієнтованої мови. Це має сенс, оскільки Python - це об’єктно-орієнтована мова.

Ітерація над списком

Для ітерації через будь-яку послідовність у JCoCo потрібен ітератор. Існують об’єкти-ітератори для кожного типу послідовностей: списків, кортежів, рядків та інших типів послідовностей, які ще не були введені. Ось програма Python, яка розділяє рядок на список рядків і перебирає список.

```
1 from disassembler import *
2 def main():
3     x = input("Enter a list: ")
4     lst = x.split()
5     for b in lst:
6         print(b)
7 disassemble(main)
```

Рис. 5.8 Збірка ітерації на списку

```
Function: main/0
Constants: None, "Enter a list: "
Locals: x, lst, b
Globals: input, split, print
BEGIN
            LOAD_GLOBAL       0
            LOAD_CONST        1
            CALL_FUNCTION     1
            STORE_FAST        0
            LOAD_FAST         0
            LOAD_ATTR         1
            CALL_FUNCTION     0
            STORE_FAST        1
            SETUP_LOOP label02
            LOAD_FAST         1
            GET_ITER
label00:    FOR_ITER label01
            STORE_FAST        2
            LOAD_GLOBAL       2
            LOAD_FAST         2
            CALL_FUNCTION     1
            POP_TOP
            JUMP_ABSOLUTE label00
label01:    POP_BLOCK
label02:    LOAD_CONST        0
            RETURN_VALUE
END
```

Рядки 6–8 коду збірки на рис. 5.8 отримують вхідний рядок від користувача, залишаючи його в стеку операндів. Рядок 9 зберігає це у змінній **`x`**. Рядки 10–12 викликають метод розділення на цьому рядку, залишаючи об’єкт списку у верхній частині стеку операндів. Список містить список рядків, розділених пробілом від вихідного рядка в **`x`**. Рядок 13 зберігає цей список у змінній **`lst`**.

Рядки 6–8 коду збірки на рис. 5.8 отримують вхідний рядок від користувача, залишаючи його в стеку операндів. Рядок 9 зберігає це у змінній **`x`**. Рядки 10–12 викликають метод розділення на цьому рядку, залишаючи об’єкт списку у верхній частині стеку операндів. Список містить список рядків, розділених пробілом від вихідного рядка в **`x`**. Рядок 13 зберігає цей список у змінній **`lst`**.

Рядок 14 встановлює точку виходу з циклу, як описано раніше в цьому розділі. Рядок 15 завантажує змінну **`lst`** у стек операндів. Інструкція **`GET_ITER`** створює ітератор у верхній частині стеку операндів. **`Lst`** вискакується зі стеку операндів під час цієї інструкції, а отриманий ітератор висувається на стек.

Ітератор має **__next__** магічний метод, який викликається інструкцією **FOR_ITER**. Коли **FOR_ITER** виконує, ітератор вискакується зі стека, на нього викликається **__next__**, а ітератор і наступне значення з послідовності висуваються в стек операндів. Ітератор залишається під наступним значенням у послідовності на **TOS1**.

Коли на ітераторі викликається **`__next__`**, і в послідовності більше не залишається елементів, ПК встановлюється на мітку інструкції **`FOR_ITER`**, що закінчує цикл. Коли цикл закінчений, стек блоків вискакує для очищення від циклу. Рядок 25 завантажує **`None`** у стек перед поверненням з основної функції.

На наступній лекції розглянемо підтримку об’єктів, функції і замикання, особливості збірки рекурсії.