Обчислення формул інтерпретацією

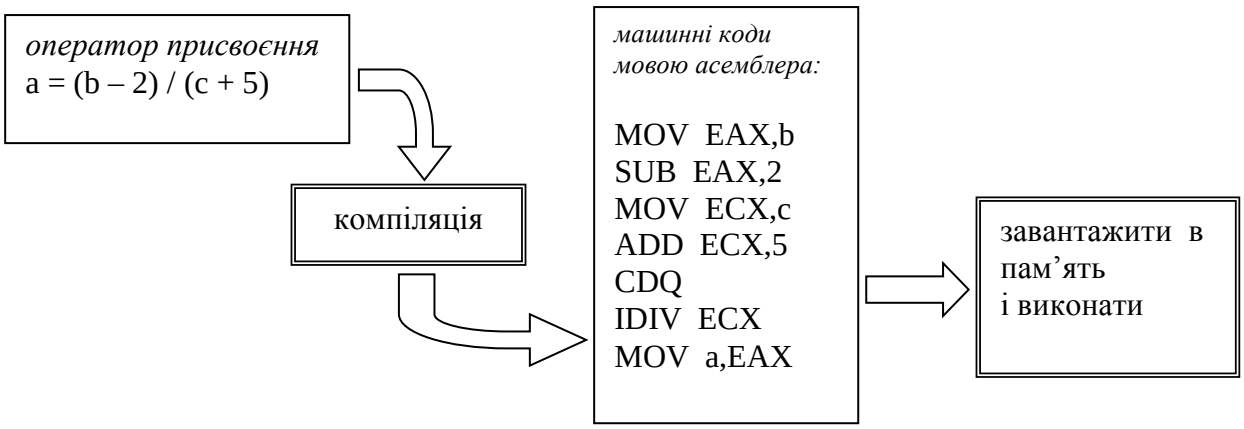
Що таке інтерпретація?

Під терміном "інтерпретація об'єкта" розуміють в загальному випадку виконання певних операцій (обчислень) над об'єктом непрямим шляхом. Непрямий шлях означає вибір алгоритму і обчислювальних операцій процесора, які би отримали такий самий результат, як пряме перетворення об'єкта до сукупності операцій процесора. Іншими словами, замість компіляції об'єкта в машинну мову обирають готові скомпільовані функції, виконання яких змоделює результат без необхідності компіляції.

Наприклад, якщо задана деяка формула у вигляді тексту:

" A = (b – 2) / (c + 5) "

тоді така формула у випадку запису в програмі компільованою алгоритмічною мовою буде перетворена компілятором в еквівалентну послідовність машинних команд:



В результаті виконання процесором машинних команд отримаємо результат обчислення, зображений у внутрішній (двійковій) формі в комірці пам'яті.

Для випадку інтерпретації формулу не компілюють в машинні команди, а виконують аналіз (сканування, синтаксичний розбір) з метою визначити дані, операції, послідовність операцій. Після цього викликають за певним алгоритмом вже готові бібліотечні функції, передають їм необхідні параметри, і отримують крок за кроком результат обчислення. Наприклад:

```
temp1 := SUB(b,2)
temp2 := ADD(c,5)
a := IDIV(temp1,temp2)
```

Тут позначення SUB(), ADD(), IDIV() є не командами процесора, записаними вище, а скомпільованими наперед підготовленими бібліотечними функціями, які викликають в потрібному порядку. Порядок виклику визначають в результаті синтаксичного розбору тексту формули.

Подібну схему інтерпретації застосовують для алгоритмічних мов, орієнтованих на реалізацію шляхом інтерпретації. Наприклад, для мови Python, яка є мовою інтерпретованого типу, маємо деяку невелику програму:

```
alst = [ 3,6,-9,35,0,0,-58,125 ]
blst = [ 25,0,-23,-9,87,35,200 ]
# порівняти списки і вибрати однакові елементи
res = [ ]
for x in alst:
  if x in blst and x not in res: res.append(x)
print(res)
# результат [-9, 35, 0]
```

Алгоритм інтерпретації може бути, наприклад, такий:

```
alst := LISTINIT(3,6,-9,35,0,0,-58,125)
blst := LISTINIT(25,0,-23,-9,87,35,200)
res := LISTINIT()
x := FIRSTOFLIST(alst)
cycle1 :
  condition1 := INLIST(x,blst);  condition2 := NOTINLIST(x,res)
  condition3 := AND(condition1,condition2)
  IF(condition3, LISTAPPEND(x,res), NONE)
  x := NEXTOFLIST(alst)
  IF(x, GOTO(cycle1), GOTO(endcycle1))
endcycle1:
PRINTLIST(res)
```

Тут великими буквами позначені знову ж таки готові бібліотечні функції, які виконують потрібні операції.

Різновиди інтерпретаторів

Інтерпретатор мови програмування (interpreter) — програма чи технічні засоби, необхідні для виконання інших програм, вид транслятора, який здійснює пооператорне (покомандне, відрядкове) опрацювання, перетворення у машинний код та виконання програми або запиту (на відміну від компілятора, який транслює у машинні коди всю програму без її виконання).

Інтерпретатори можуть працювати як з початковим кодом програми (source code), написаним мовою програмування, так і з байт-кодом (інтерпретатори байт-коду).

Простий інтерпретатор аналізує і відразу виконує (власне інтерпретація) програму покомандно (або відрядково), по мірі надходження тексту програми на вхід інтерпретатора. Перевагою такого підходу є миттєва реакція. Недолік — такий інтерпретатор виявляє помилки в тексті програми тільки при спробі виконання команди (або рядка) з помилкою.

Інтерпретатор компілюючого типу — це система з компілятора, який перекладає текст програми в проміжне представлення, наприклад, в байт-код або p-код, і власне інтерпретатора, який виконує отриманий проміжний код (так звана віртуальна машина). Перевагою таких систем є більша швидкодія виконання програм (за рахунок винесення аналізу початкового коду в окремий, разовий прохід, і мінімізації цього аналізу в інтерпретаторі). Недоліки — більші вимоги до ресурсів і вимога на коректність тексту програми. Застосовується в таких мовах, як Java, Tcl, Perl (використовується байт-код), REXX (зберігається результат синтаксичного аналізу), а також у різних СУБД (використовується p-код).

Інтерпретатор компілюючого типу складається з компілятора мови і простого інтерпретатора з мінімізованим аналізом початкового коду. Цей код в такому випадку не обов'язково повинен мати текстовий формат — це може бути машинний код якоїсь наявної апаратної платформи. Наприклад, віртуальні машини типу QEMU, Bochs, VMware включають в себе інтерпретатори машинного коду процесорів сімейства x86.

Деякі інтерпретатори (наприклад, для мов Lisp, Scheme, Python, Basic та інших) можуть працювати в режимі діалогу або так званого циклу читання-обчислення-друку (англ. read-eval-print loop, REPL). У такому режимі інтерпретатор зчитує закінчену конструкцію мови (наприклад, s-expression у мові Lisp), виконує її, друкує результати, після чого переходить до очікування введення користувачем наступної конструкції.

Проміжний код

Проміжний код для Java. Початковий код на Java компілюється в проміжний код, який буде інтерпретовано. Проміжною мовою для Java є байт-код, інтерпретатор — Java Virtual Machine (JVM). Файл байт-коду є універсальним, тоді як інтерпретатор є унікальним для кожної платформи.

Розглянемо такий приклад на мові Java:

```
outer:
  for (int i = 2; i < 1000; i++) {
      for (int j = 2; j < i; j++) {
          if (i % j == 0)
              continue outer;
      }
      System.out.println (i);
  }
```

Компілятор Java може транслювати цей приклад в такий байт-код:

```
  0:   iconst_2
  1:   istore_1
  2:   iload_1
  3:   sipush  1000
  6:   if_icmpge       44
  9:   iconst_2
  10:  istore_2
  11:  iload_2
  12:  iload_1
  13:  if_icmpge       31
  16:  iload_1
  17:  iload_2
  18:  irem
  19:  ifne    25
  22:  goto    38
  25:  iinc    2, 1
  28:  goto    11
  31:  getstatic #84; //Field java/lang/System.out:Ljava/io/PrintStream;
  34:  iload_1
  35:  invokevirtual  #85; //Method java/io/PrintStream.println:(I)V
  38:  iinc    1, 1
  41:  goto    2
  44:  return
```

Проміжний код для мов .NET Framework. Мови, сумісні з платформою .NET Framework, такі як Visual Basic, C#, Visual F#, VB.NET, J#, компілюються в Common Intermediate Language (CIL) — проміжну мову для платформи .NET Framework. Отриманий проміжний код інтерпретується Common Language Runtime (CLR) — загальномовним середовищем виконання. Специфікація CIL та CLR є реалізацією специфікації Common Language Infrastructure (CLI) — специфікації загальномовної інфраструктури компанії Microsoft.

Код на CIL генерують всі компілятори для платформи .NET Framework. Мова CIL по структурі та мнемоніці нагадує мову асемблера. Проте CIL містить деякі високорівневі конструкції, і писати на CIL значно легше, ніж на асемблері.

Приклад програми на C#:

```
static void Main(string[] args)
{
outer:
  for (int i = 2; i < 1000; i++)
  {
     for (int j = 2; j < i; j++)
     {
       if (i % j == 0)  goto outer;
     }
     Console.WriteLine(i);
  }
}
```

Вигляд цієї ж програми на CIL:

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  .maxstack  2
  .locals init ([0] int32 i,
           [1] int32 j)
  IL_0000:  ldc.i4.2
            stloc.0
            br.s       IL_001f
  IL_0004:  ldc.i4.2
            stloc.1
            br.s       IL_0011
  IL_0008:  ldloc.0
            ldloc.1
            rem
            brfalse.s  IL_0000
            ldloc.1
            ldc.i4.1
            add
            stloc.1
  IL_0011:  ldloc.1
            ldloc.0
            blt.s      IL_0008
            ldloc.0
            call       void [mscorlib]System.Console::WriteLine(int32)
            ldloc.0
            ldc.i4.1
            add
            stloc.0
  IL_001f:  ldloc.0
            ldc.i4     0x3e8
            blt.s      IL_0004
            ret
}
```

Правила будови синтаксичних визначень

Для точного визначення будови деякої синтаксичної конструкції приймемо такі позначення. Їх використання є загальноприйнятим, і має основою формули БНФ:
- *нетермінали* позначаємо звичайним ідентифікатором, наприклад `number`;
- *термінали* позначаємо лапками, наприклад `"+", "(", "sqrt"`;
- *круглі дужки* використовуємо подібно, як в алгебрі, для групування інших конструкцій, наприклад ("+" | "-" | "*" | "/"); самі дужки в такому записі не є частиною конструкції;
- *вертикальна риска* | означає "або", тобто вибір однієї з альтернатив;
- *знак зірочки* * означає повторення нуль або більше разів конструкції, записаної перед зірочкою, наприклад `cipher *` ; сама зірочка не є частиною конструкції;
- *квадратні дужки* [ ] означають або відсутність, або однократне входження в синтаксичний запис, наприклад `[ prompt ]` ;
- *знак* ::= читаємо "за означенням є".

Найпростіший приклад інтерпретації

Задача. В текстовому файлі чи в текстовому рядку записана послідовність десяткових чисел і знаків арифметичних операцій (формула). Наприклад:

65 + 122 – 99 / 6 – 12 * 2 + 1

Необхідно виконати обчислення формули за такими правилами і обмеженнями: числа десяткові цілі; допустимі операції є +, -, *, /; операції рангу не мають і їх виконують зліва направо в порядку запису; результатом кожної операції є ціле або дійсне число; формула записана коректно, зокрема, для запису чисел використані лише десяткові цифри і немає операції ділення на нуль. Така формула дуже нагадує просту послідовність операцій на звичайному калькуляторі.

Спочатку треба дати точне синтаксичне означення правил запису формули:

```
arith_expr  ::= number  ( operator  number ) *
number  ::=  cipher  cipher *
cipher  ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
operator  ::=   "+"  |  "-"  |  "*"  |  "/"
```

Зауважимо, що нетермінал cipher часто вважають терміналом "cipher", розуміючи будь-яку десяткову цифру – для скорочення визначень.

Отже, формула може мати одне число або більше, розділених знаками арифметичних операцій. Запис кожного числа є нерозривний. Між числами і знаками можуть бути символи пропуску, вони є незначущими і їх не включають в синтаксичні правила. Подібні символи зазвичай викреслюють з тексту ще до початку переглядів.

Далі подано модельний приклад інтерпретатора, записаний мовою Python. Інтерпретатор працює за два перегляди тексту формули. До початку першого перегляду метод `delblank()` викреслює з тексту всі незначущі пропуски. За першим переглядом будують список лексем цілої формули. За другим переглядом обчислюють формулу, якщо не було помилок за першим переглядом. У випадку помилок друкують найпершу помилкову знайдену літеру.

Оператори `print()` метода `calc()` можна розкоментувати, щоб побачити проміжні результати процедури обчислення. Можна також додати для експериментів інші оператори `print()` до тексту методів, і отримати спостереження за ходом виконання інтерпретатора.

Програмний код інтерпретатора є в файлі `simple.py`.

```
# простий приклад інтерпретації
# перший перегляд - сканування формули і будова списку лексем
# другий перегляд - обчислення формули
class SimpleInterpret:

  def __init__(self,text): # конструктор
    self.text=text  # копія тексту формули
    self.leks = [] # список лексем
    self.i = 0 # поточна позиція сканування літери

  def calc(self): # виконати повну процедуру обчислення
    self.delblank()  # викреслити пропуски
    if not self.scanner():  # перший перегляд - сканування формули
      return (False,self.text[self.i]) # помилки сканування - літера
    else:  # другий перегляд - обчислення формули
      #print(self.leks)  # контроль списку лексем
      # додати на початок списку лексем знак "+",
      #   буде простіший алгоритм:
      self.leks.insert(0,"+")
      #print(self.leks)  # контроль піля insert(0,"+")
      k = 0  # поточний номер лексеми
      res = 0  # результат обчислення
      while k < len(self.leks) : # пари [ знак,операнд ]
        oper = self.leks[k];  k+=1  # знак
        n = int(self.leks[k]);  k+=1  # числа вважаємо цілими
        res = res+n if oper=='+' else res-n if oper=='-'   \
              else res*n if oper=='*' else res/n
      return (True,res)

  def delblank(self): # викреслити пропуски - незначущі літери
    self.text = self.text.replace(' ','')

  def scanner(self): # сканувати формулу і поділити на лексеми
    n = self.onenumber() # на першій позиції - число
    if n != None: self.leks.append(n)  # правило  formula::=number
    else: return None # помилка в найпершому числі
    while self.i<len(self.text): # правило formula::=(operator number) *
      sign = self.onesign() # читати знак
      if sign != None: self.leks.append(sign)
      else: return None # помилка знаку операції
      n = self.onenumber() # наступна позиція - число
      if n != None: self.leks.append(n)
      else: return None # помилка в числі
    return "OK" # всі лексеми правильні

  def onenumber(self):
    # читати літери числа - правило  number::= cipher  cipher *
    num = ""
    while self.i < len(self.text) and self.text[self.i].isdigit():
      num += self.text[self.i];  self.i+=1
    if len(num) > 0: return num
    else: return None

  def onesign(self):
    # читати знак операції - правило  operator::= "+" | "-" | "*" | "/"
    if self.text[self.i] in [ '+', '-', '*', '/' ]:
      self.i+=1;  return self.text[self.i-1]
    else: return None
```

```
if __name__ == "__main__" :
  formula = "65 + 122 - 99 / 6 - 12 * 2 + 1"  # задана формула
  #formula = "65 + A * 4"
  res = SimpleInterpret(formula).calc()
  if res[0]:  print(res[1])
  else:
    print("Error symbol:", res[1])
```

Для першої заданої формули отримаємо результат 6.333333333333332.

Якщо розкоментувати оператор

```
#print(self.leks)  # контроль піля insert(0,"+")
```

то побачимо такий список лексем, як доповнений результат першого перегляду тексту:

```
['+', '65', '+', '122', '-', '99', '/', '6', '-', '12', '*', '2', '+', '1']
```

Для другої формули отримаємо результат `Error symbol: A`

Отже, підсумково кроки виконання інтерпретації є такі:

задано	`"65 + 122 - 99 / 6 - 12 * 2 + 1"`
викреслити пропуски	`"65+122-99/6-12*2+1"`
поділити на лексеми	`['65', '+', '122', '-', '99', '/', '6', '-', '12', '*', '2', '+', '1']`
додати знак "+"	`['+', '65', '+', '122', '-', '99', '/', '6', '-', '12', '*', '2', '+', '1']`
цикл обчислення пар [знак,операнд]	`['+', '65'], ['+', '122'], ['-', '99'], ['/', '6'], ['-', '12'], ['*', '2'], ['+', '1']`

.

Інтерпретація формул загальних алгебраїчних правил

Розглянемо тепер записи формул, виконані відомими алгебраїчними правилами Операції поділені на ранги: 1-й ранг "множення" і "ділення", другий ранг "додавання" і віднімання". Будемо для початку вважати, що інших операцій немає. Запис круглих дужок означає зміну порядку виконання операцій. Операції однакового рангу виконують зліва направо.

Для першого вивчення розглядаємо формули, які складаються лише з цілих чисел, наприклад:

49 – 108 / (6 + 11) * (100 – 94)

Так само, як в попередньому розділі, треба дати точне синтаксичне означення правил запису формули. Але тепер потрібно врахувати загальні алгебраїчні правила:

```
arith_expr ::=  term  ( ( "+" | "-" )  term ) *
term ::=  factor  ( ( "*" | "/" )  factor ) *
factor ::=   number  |  "(" arith_expr ")"
number  ::=  cipher  cipher *
cipher  ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Аксіомою цієї граматики є `arith_expr`. З точки зору правил граматичного розбору і будови виведень в граматиках приорітет буде зростати при переході до правила, яке використовують для підстановки. Наприклад, якщо маємо 20+10*4, тоді виведення виглядає так в лівосторонньому порядку (для спрощення запису лапки не показуємо, деякі підстановки об'єднуємо):

arith_expr => term + term => factor + term => number + term => 20 + term =>
=> 20 + factor * factor => 20 + number * factor => 20 + 10 * factor => 20 + 10 * number =>
=> 20 + 10 * 4

Операції будуть виконані в оберненому порядку до кроків виведення. Спочатку буде виконана операція множення, після неї – операція додавання. Операцію можна виконати лише тоді, коли обидва операнди обчислені.

Реалізацію граматики виразів і розпізнавання виразу реалізуємо методом нисхідного граматичного розбору за *алгоритмом рекурсивного спуску*. З теорії формальних мов і граматик відомо, що для реалізації методу рекурсивного спуску потрібно переписати граматичні правила у вигляді, де немає ліворекурсивних правил. Для нашого випадку ця вимога виконана.

За методом рекурсивного спуску до кожного нетермінального символа будують окрему функцію, яка повинна розпізнати праву частину граматичного правила. При цьому функція може викликати в потрібні моменти інші функції для розпізнавання частини конструкції. Наприклад, функція для `arithexpr` буде викликати функцію для `term`, функція для `term` буде викликати функцію для `factor`, і т.д. Врешті за такою схемою можливі непрямі рекурсії, наприклад, `arithexpr` → `term` → `factor` → `(arithexpr)` → … .

Вхідними даними для програми обчислення виразу є масив лексем, побудований сканером. Арифметичний вираз може мати довільний вигляд в межах записаних вище синтаксичних правил. Тому доцільно підготувати список лексем в формі, зручній для синтаксичного розбору. Кожну лексему можна подати у вигляді (код,значення), де код – числове позначення лексеми кожного виду. Приймемо такі позначення лексем:

number	1	значення числа
openbracket	3	"("
closebracket	4	")"
add	5	"+"
subtract	6	"-"

multiply	7	"*"
divide	8	"/"

Список лексем легко отримати, переглядаючи синтаксичні правила визначення формули.

Програмна реалізація (файл `arithexpr.py`)

```
# Інтерпретація формул загальних алгебраїчних правил

# допоміжний клас для збудження винятків
class errorexpr(Exception): pass  # можливі помилки

class ArithexprInterpret:

  # common data 1
  empty = 0
  number =1
  openbracket = 3
  closebracket = 4
  add = 5
  subtract = 6
  multiply = 7
  divide = 8

  # common data 2
  errscan = 'Недопустима літера в тексті формули:\n'
  errcalc = 'Помилка обчислення виразу:\n'

  def __init__(self,text): # конструктор
    self.text=text  # копія тексту формули
    self.leks = [] # список лексем
    self.i = 0 # поточна позиція сканування літери
    self.k = 0 # поточна позиція лексеми при обчисленні

  def calc(self): # виконати повну процедуру обчислення
    self.delblank()  # викреслити пропуски
    #print(self.text)
    if not self.scanner():  # перший перегляд - сканування формули
      return (False,self.errscan + self.text[:self.i] + "  '" +
              self.text[self.i] + "'")  # помилки сканування
    #print(self.leks)
    # другий перегляд - розбір і обчислення формули
    res = None
    try:
      res = self.arithexpr()  # запуск розбору
    except : # errorexpr та інші   # були помилки
      temp = "".join(map(lambda m: str(m[1]),self.leks[:self.k]))
      return (False,self.errcalc + temp + "  '" +  \
              str(self.leks[self.k][1]) + "'")
    if self.k < len(self.leks)-1 : # не враховано останню частину виразу
      temp = "".join(map(lambda m: str(m[1]),self.leks[:self.k]))
      return (False,self.errcalc + temp + "  '" +  \
              str(self.leks[self.k][1]) + "'")
    return (True,res)
```

```
  def delblank(self): # викреслити пропуски - незначущі літери
    self.text = self.text.replace(' ','')

  def scanner(self): # сканувати формулу і поділити на лексеми
    while self.i < len(self.text) :
      if self.text[self.i] == '(' :
        self.leks.append( (self.openbracket,'(') )
      elif self.text[self.i] == ')' :
        self.leks.append( (self.closebracket,')') )
      elif self.text[self.i] == '+' : self.leks.append( (self.add,'+') )
      elif self.text[self.i] == '-' :
        self.leks.append( (self.subtract,'-') )
      elif self.text[self.i] == '*' :
        self.leks.append( (self.multiply,'*') )
      elif self.text[self.i] == '/' :
        self.leks.append( (self.divide,'/') )
      elif self.text[self.i].isdigit(): self.onenumber(); self.i -=1
      else: return False # недопустима літера
      self.i += 1
    self.leks.append( (self.empty,'#') ) # обмежувач списку лексем
    return True

  def onenumber(self):
    # читати літери числа - правило  number::= cipher  cipher *
    num = ""
    while self.i < len(self.text) and self.text[self.i].isdigit():
      num += self.text[self.i];  self.i+=1
    if len(num) > 0: self.leks.append( (self.number,int(num)) )
    else: return None

  def arithexpr(self):
    y = self.term()  # найперший доданок: правило  arith_expr ::= term
    while self.leks[self.k][0]==self.add or
          self.leks[self.k][0]==self.subtract:
      # наступні доданки: правило arith_expr::= ( ( "+" | "-" ) term ) *
      opr = self.leks[self.k][0]  # запам'ятати операцію
      self.GetNextToken()  # перейти до наступної лексеми
      if opr==self.add : y = y + self.term()
      else : y = y - self.term()
    return y

  def term(self):
    z = self.factor()  # найперший множник: правило term ::= factor
    while self.leks[self.k][0]==self.multiply or
          self.leks[self.k][0]==self.divide:
      # наступні множники: правило  term::= ( ( "*" | "/" ) factor ) *
      opr = self.leks[self.k][0]  # запам'ятати операцію
      self.GetNextToken()  # перейти до наступної лексеми
      if opr == self.multiply : z = z * self.factor()
      else : z = z / self.factor()
    return z

  def factor(self):
    if self.leks[self.k][0]==self.number : # правило  factor ::= number
      self.GetNextToken()  # перейти до наступної лексеми
      return self.leks[self.k-1][1] # повернути число попередньої лексеми
    elif  self.leks[self.k][0]==self.openbracket :
      # правило  factor ::=  "(" arith_expr ")"
```

```
      self.GetNextToken()  # перейти до наступної лексеми
      ex = self.arithexpr()  # частина виразу в дужках
      if self.leks[self.k][0]==self.closebracket :  self.GetNextToken()
      else: raise errorexpr  # ? немає закриваючої дужки
      return ex
    else : return None

  def GetNextToken(self):  # перейти до наступної лексеми
    if self.k < len(self.leks)-1: self.k+=1
    else: raise errorexpr  # ? неможливо продовжити аналіз


if __name__ == "__main__" :
  formula = "49 - 108 / (6 + 11) * (100 - 94)"
  res = ArithexprInterpret(formula).calc()
  if res[0]:  print(res[1])
  else:
    print("Error :", res[1])
```

Зробимо важливі зауваження. 1) Як було сказано вище, кожна лексема списку є парою (код,значення), що дозволяє спростити перегляд лексем лише за кодами, а значення – за потреби. 2) В кінець списку лексем додаємо в методі `scanner(self)` обмежувач списку лексем `(self.empty,'#')`. Це дозволяє уникнути частих перевірок вичерпання списку лексем в кожному методі. У випадку досягнення обмежувача і спроби продовжити аналіз відбудеться генерування помилки і автоматичне припинення процедури розбору і обчислення. 3) Метод `GetNextToken(self)` виконує контролюваний перехід до наступної лексеми і генерування помилки у випадку необхідності. 4) Методи розбору і обчислення `arithexpr()`, `term()`, `factor()` і `GetNextToken()` мають бути дуже точно синхронізовані щодо пересування списком лексем. Перед викликом кожного з методів `arithexpr()`, `term()`, `factor()` черговою лексемою для аналізу має бути найперша лексема відповідної частини арифметичного виразу.

Рекомендуємо виконати обчислювальні експерименти з інтерпретатором формул. По-перше, можна розкоментувати в методі `calc()` оператори `print()`, щоб побачити кроки роботи. Наприклад, для показаної вище формули

$$49 – 108 / (6 + 11) * (100 – 94)$$

отримаємо такі результати:

```
49-108/(6+11)*(100-94)
[(1, 49), (6, '-'), (1, 108), (8, '/'), (3, '('), (1, 6), (5, '+'),
 (1, 11), (4, ')'), (7, '*'), (3, '('), (1, 100), (6, '-'), (1, 94),
 (4, ')'), (0, '#')]
10.882352941176471
```

По-друге, варто записати формулу з синтаксичними помилками і переглянути повідомлення інтерпретатора для випадків помилок.

По-третє, можна додати для експериментів інші оператори `print()` до тексту методів, і отримати спостереження за ходом виконання інтерпретатора.

Розширення змісту формул

Маючи записані раніше правила для арифметичних формул, їх можна розширювати для отримання додаткових можливостей. Наприклад, ми хочемо мати ще таке:
1) бінарні операції ділення на ціло //, остача від ділення %;
2) піднесення до степеня **; звернемо увагу, що піднесення до степеня має вищий ранг, ніж операції множення і ділення;
3) додати функцію sin(x);
4) унарні операції + і -; можна записати перед будь-яким співмножником чи доданком; мають вищий ранг від бінарних операцій *, /, +, -, але нижчий від операції піднесення до степеня **; наприклад -2+-6, 4*+8; отже, якщо записані два знаки операцій підряд, тоді перший знак означає бінарну операцію, а другий – унарну;
5) крім цілих чисел дозволити дійсні числа фіксованої крапки і в експоненціальній формі, наприклад 45.02, 1.0e-5, 28e4; зафіксуємо вимогу, що запис будь-якого числа завжди починається з цифри.

Отже, треба змінити правила так, щоб зберегти раніше визначену частину правил, додати нові операції і врахувати приорітети нових операцій.

Це можна зробити, наприклад, так:

```
arith_expr ::=  term  ( ( "+" | "-" )  term ) *
  # term ::=  factor  ( ( "*" | "/" )  factor ) *    # було
term ::=  factor  ( ( "*" | "/" | "//" | "%" )  factor ) *
  # factor ::=  number  |  "(" arith_expr ")"       # було
factor ::= [( + | - )] ( number | "(" arith_expr ")" | function )
             [( "**" factor )*]
number ::=  cipher  cipher * [ . cipher cipher * ]
               [ (e|E) [(+|-)] cipher cipher * ]
function ::=  "sin(" arith_expr ")"
cipher ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Множником (`factor`) тепер вважаємо елемент (число, вираз в дужках, функція), який може мати операцію `**` піднесення до степеня. Показник степеня знову трактуємо як множник. Зауважимо, що при такому визначенні декілька підряд записаних операцій піднесення до степеня без дужок будуть виконані справа наліво:

2 ** 3 ** 4 означає 2 ** (3 ** 4), але не (2 ** 3) ** 4

Зрештою, таке трактування послідовних піднесень до степеня прийнято в багатьох алгоритмічних мовах.

Крім того, додаємо можливість унарної операції + або – перед множником.

Перелік можливих змін є такий:

1) доповнити сканер новими лексемами;

2) додати до метода `term()` нові бінарні операції;

3) змінивши правило для `factor`, тепер треба відповідно переписати метод `factor()`. Для цього можна повторювати в циклі (або рекурсивно) основну частину записаної раніше функції і в цій же функції обчислювати піднесення до степеня;

4) треба інакше визначити метод `onenumber()`, щоб він дозволяв форму цілого або дійсного;

5) варто визначити окремий метод `funcname()` для сканера, щоб відділити ім'я функції sin;

6) до метода `factor()` додати можливість унарної операції + або -.

Розширений варіант інтерпретатора формул є в файлі `extarithexpr.py`.

Операторне виконання

Щоб інтерпретація мала закінчену форму, потрібно ще раз доповнити граматику – до операторної форми виконання. Отже, будемо мати не лише формули самі по собі, але оператори присвоєння результатів обчислень. Присвоєння виконуємо змінним величинам, які можуть бути довільними ідентифікаторами. Проте зараз для спрощення дозволимо для ідентифікаторів лише латинські букви, інших знаків використовувати не будемо. Додамо також можливість використовувати в формулах не лише константи, але й змінні величини.

Крім того, щоб бачити результати обчислень вже під час виконання, введемо оператор `print()` друкування змінних величин.

Всі змінні величини та їх поточні значення будемо зберігати в словнику `dict`, використовуючи вбудовані правила словника. А саме: присвоєння, наприклад, `D['test']=95` приводить до додавання нового ключа `'test'` (змінної величини) і значення, якщо такого ключа ще немає, а якщо ключ (змінна величина) вже є – тоді фіксується нове значення. В такий спосіб маємо автоматичне стеження за переліком змінних величин.

Отже, якщо в формулі (права частина оператора присвоєння) чи операторі друкування є посилання на змінну величину, тоді спочатку перевіряємо наявність такої величини в словнику. Якщо є – отримуємо з словника її значення. Якщо немає – тоді це помилка обчислення "звертання до неіснуючої змінної".

Повна граматика нашої мікромови буде така:

```
prog ::=  ( assign NEWLINE | print NEWLINE ) +   # нове правило
assign ::=  var "=" arith_expr          # нове
print ::=  "print(" varlist ")"         # нове
varlist ::= var ( "," var ) *           # нове
var ::= "letter"  "letter" *            # нове

arith_expr ::=  term  ( ( "+" | "-" )  term ) *
term ::=  factor  ( ( "*" | "/" | "//" | "%" )  factor ) *
factor ::= [( + | - )] ( number | "(" arith_expr ")" | function | var )
             [( "**" factor )*]         # додали змінну var
number ::=  cipher  cipher * [ . cipher cipher * ]
               [ (e|E) [(+|-)] cipher cipher * ]
function ::=  "sin(" arith_expr ")"
cipher ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Зауваження щодо запису граматичних правил. Знак + (плюс) означає повторення конструкції один або більше разів, на відміну від знака *, для якого повторення могло бути нуль разів. Терміналом `"letter"` позначаємо будь-яку велику або малу латинську букву – щоб на робити довгих перелічень всіх букв.

Отже, програмою `prog` є довільна послідовність операторів присвоєння і друкування. Кожен оператор має бути записаний окремим рядком, який закінчується переходом до наступного радка `NEWLINE`.

Приклад програми:

```
alfa = -10 + 2 * sin(2/10 - sin(1)) + 5
beta = 256
x = beta - alfa * 150
print(alfa,beta,x)
y = 2 * x - alfa ** 2
print(y)
```

Інтерпретатор для операторного виконання є в файлі `extarithexpr-ass.py`.