

Урок по синтаксису языка Ukuvchi

С знака \$ начинается выполнение операторов слева направо . Например надо объявить переменную a (в языке используются только диапазон букв a – z , где z регистр куда записывать возвращаемое значение функцией)

```
(  
  ($  
    (set! a (100))  
  )  
)
```

Или это можно сделать так

```
(  
  ($  
    (set! a (arif 100))  
  )  
)
```

Для арифметических операций поддерживаются следующие символы :

^ - возведение в степень

* - умножение

/ - деление

% - остаток от деления

+ - сложение

- - вычитание

Приоритет операций осуществляется не круглыми скобками , а квадратными []. Между символами оператора и чисел должен быть минимум один пробел .

Можем просто записать арифметическое выражение , не занося результат в определенную переменную и посмотреть на стек виртуальной машины :

```
(  
  ($  
    (arif [ 12 + 6 ] * 3 )  
  )  
)
```

Виртуальная машина должна показать такие данные :

```
[ 'arif', '[', 12.0, '+', 6.0, ']', '*', 3.0 ]
```

```
func_table: {}
mas_I_Or_Str:
vector<int>_b_c:[12, 65, 64, 0, 0, 12, 64, 192, 0, 0, 1, 12, 64, 64, 0, 0, 3, 23]
start_ip:0
0000: ICONST 12.000000stack=[ 12.000000 ]
0005: ICONST 6.000000stack=[ 12.000000 6.000000 ]
0010: iadd          stack=[ 18.000000 ]
0011: ICONST 3.000000stack=[ 18.000000 3.000000 ]
0016: imul          stack=[ 54.000000 ]
```

Занесем значение выражение в переменную c и отпечатаем ее :

```
( $
(set! c ( arif [ 12 + 6 ] * 3 ))
(print c)
)
```

Виртуальная машина должна показать такое :

```
['set!', 'c', ['arif', ['12.0', '+', '6.0', ''], '*'], 3.0]]
['arif', ['12.0', '+', '6.0', ''], '*'], 3.0]
['print', 'c']
func_table: {}
mas_I_Or_Str:
vector<int>_b_c:[12, 65, 64, 0, 0, 12, 64, 192, 0, 0, 1, 12, 64, 64, 0, 0, 3, 15, 2,
17, 2, 23]
start_ip:0
0000: ICONST 12.000000stack=[ 12.000000 ]
0005: ICONST 6.000000stack=[ 12.000000 6.000000 ]
0010: iadd          stack=[ 18.000000 ]
0011: ICONST 3.000000stack=[ 18.000000 3.000000 ]
0016: imul          stack=[ 54.000000 ]
0017: store    2      stack=[ ]
0019: print          print: 54.000000
```

(Здесь есть и вывод компилятора)

Условия

Есть комментарии , пишем так (*// комментарий*) и есть оператор *pass* , который означает ничего не делать . Итак условия , сравним 10 и 10.1 на меньше , если $10 < 10.1$ присвоим переменной *x* значение 100 и выведем *print* :

```
(  
  (// Пример if с else как pass)  
  (if(< (arif 10) (arif 10.1))  
    ($ (set! x (arif 100)) (// Последовательность True ветки)  
      (print x)           (// Ожидается 100))  
    (pass))  
)
```

Вывод ВМ :

```
start_ip:0  
0000: ICONST 10.000000stack=[ 10.000000 ]  
0005: ICONST 10.100000stack=[ 10.000000 10.100000 ]  
0010: ilt      stack=[ 1.000000 ]  
0011: brf     24      stack=[ ]  
0013: ICONST 100.000000stack=[ 100.000000 ]  
0018: store   23      stack=[ ]  
0020: print           print: 100.000000  
stack=[ ]  
0022: br      25      stack=[ ]
```

Условия с *else* :

```
(  
  (// Пример if с else )  
  (if(< (arif 20) (arif 10))  
    ($ (set! x (arif 100)) (// Последовательность True ветки)  
      (print x))  
    ($ (set! x (arif 200)) (// Последовательность False ветки)  
      (print x)           (// Ожидается 200)))  
)
```

Вывод ВМ :

```
start_ip:0
```

```

0000: ICONST 20.000000stack=[ 20.000000 ]
0005: ICONST 10.000000stack=[ 20.000000 10.000000 ]
0010: ilt      stack=[ 0.000000 ]
0011: brf     24      stack=[ ]
0024: ICONST 200.000000stack=[ 200.000000 ]
0029: store   23      stack=[ ]
0031: print      print: 200.000000
stack=[ ]

```

Пример на равенство :

```

($
  (// Пример if с else как pass - равенство)
  (if(= (arif 10.2) (arif 10.2))
    ($ (set! x (arif 100)) (// Последовательность True ветки)
      (print x)           (// Ожидается 100))
    (pass))
  )

```

Вывод ВМ

```

start_ip:0
0000: ICONST 10.200000stack=[ 10.200000 ]
0005: ICONST 10.200000stack=[ 10.200000 10.200000 ]
0010: ieq      stack=[ 1.000000 ]
0011: brf     24      stack=[ ]
0013: ICONST 100.000000stack=[ 100.000000 ]
0018: store   23      stack=[ ]
0020: print      print: 100.000000
stack=[ ]
0022: br      25      stack=[ ]

```

Если числа не равны :

```

($
  (// Пример if с else - равенство)
  (if(= (arif 10.2) (arif 10.1))
    ($ (set! x (arif 100)) (// Последовательность True ветки)
      (print x)           )
    )
  )

```

```
($ (set! x (arif 200)) (// Последовательность False ветки)
(print x)           (// Ожидается 200)))
)
```

Вывод ВМ :

```
start_ip:0
0000: ICONST 10.200000stack=[ 10.200000 ]
0005: ICONST 10.100000stack=[ 10.200000 10.100000 ]
0010: ieq      stack=[ 0.000000 ]
0011: brf     24      stack=[ ]
0024: ICONST 200.000000stack=[ 200.000000 ]
0029: store   23      stack=[ ]
0031: print          print: 200.000000
stack=[ ]
```

Хочу сказать , если в синтаксисе такого lisp – подобного языка ставить скобки некоего логического выражения на новой строке с соответствующими отступами , то соштрится вполне высокоуровнево :)

Цикл while

У while такой шаблон :

```
( while ( <test> ) ( [$] body ) )
```

Запишем такой алгоритм , который на псевдо-языке будет выглядеть так :

```
x:=1 + 2 * 3 // 7
```

```
i:=0
```

```
while ( i < x ) :
```

```
    print ( x )
```

```
    i +=1
```

На Ukuvchi так :

```

($
  (// Пример цикл while)
  (set! x (arif 1 + 2 * 3))
  (set! i 0)
  (while
    (< (arif i) (arif x) )
    ($ (print i) (set! i (arif i + 1)) )
  )
)

```

Вывод ВМ :

```

start_ip:0
0000: ICONST 1.000000stack=[ 1.000000 ]
0005: ICONST 2.000000stack=[ 1.000000 2.000000 ]
0010: ICONST 3.000000stack=[ 1.000000 2.000000 3.000000 ]
0015: imul      stack=[ 1.000000 6.000000 ]
0016: iadd      stack=[ 7.000000 ]
0017: store 23   stack=[ ]
0019: ICONST 0.000000stack=[ 0.000000 ]
0024: store 8    stack=[ ]
0026: load 8     stack=[ 0.000000 ]
0028: load 23    stack=[ 0.000000 7.000000 ]
0030: ilt       stack=[ 1.000000 ]
0031: brf 47     stack=[ ]
0033: print      print: 0.000000
stack=[ ]
0035: load 8     stack=[ 0.000000 ]
0037: ICONST 1.000000stack=[ 0.000000 1.000000 ]
0042: iadd      stack=[ 1.000000 ]
0043: store 8    stack=[ ]
0045: br 26     stack=[ ]
0026: load 8     stack=[ 1.000000 ]
0028: load 23    stack=[ 1.000000 7.000000 ]
0030: ilt       stack=[ 1.000000 ]
0031: brf 47     stack=[ ]
0033: print      print: 1.000000
stack=[ ]
0035: load 8     stack=[ 1.000000 ]
0037: ICONST 1.000000stack=[ 1.000000 1.000000 ]

```

```

0042: iadd      stack=[ 2.000000 ]
0043: store      8      stack=[ ]
0045: br         26      stack=[ ]
0026: load        8      stack=[ 2.000000 ]
0028: load       23      stack=[ 2.000000 7.000000 ]
0030: ilt          stack=[ 1.000000 ]
0031: brf         47      stack=[ ]
0033: print          print: 2.000000
stack=[ ]
0035: load        8      stack=[ 2.000000 ]
0037: ICONST 1.000000 stack=[ 2.000000 1.000000 ]
0042: iadd      stack=[ 3.000000 ]
0043: store      8      stack=[ ]
0045: br         26      stack=[ ]
0026: load        8      stack=[ 3.000000 ]
0028: load       23      stack=[ 3.000000 7.000000 ]
0030: ilt          stack=[ 1.000000 ]
0031: brf         47      stack=[ ]
0033: print          print: 3.000000
stack=[ ]
0035: load        8      stack=[ 3.000000 ]
0037: ICONST 1.000000 stack=[ 3.000000 1.000000 ]
0042: iadd      stack=[ 4.000000 ]
0043: store      8      stack=[ ]
0045: br         26      stack=[ ]
0026: load        8      stack=[ 4.000000 ]
0028: load       23      stack=[ 4.000000 7.000000 ]
0030: ilt          stack=[ 1.000000 ]
0031: brf         47      stack=[ ]
0033: print          print: 4.000000
stack=[ ]
0035: load        8      stack=[ 4.000000 ]
0037: ICONST 1.000000 stack=[ 4.000000 1.000000 ]
0042: iadd      stack=[ 5.000000 ]
0043: store      8      stack=[ ]
0045: br         26      stack=[ ]
0026: load        8      stack=[ 5.000000 ]
0028: load       23      stack=[ 5.000000 7.000000 ]
0030: ilt          stack=[ 1.000000 ]
0031: brf         47      stack=[ ]

```

```

0033: print      print: 5.000000
stack=[ ]
0035: load      8      stack=[ 5.000000 ]
0037: ICONST 1.000000stack=[ 5.000000 1.000000 ]
0042: iadd      stack=[ 6.000000 ]
0043: store     8      stack=[ ]
0045: br       26      stack=[ ]
0026: load      8      stack=[ 6.000000 ]
0028: load     23      stack=[ 6.000000 7.000000 ]
0030: ilt       stack=[ 1.000000 ]
0031: brf      47      stack=[ ]
0033: print      print: 6.000000
stack=[ ]
0035: load      8      stack=[ 6.000000 ]
0037: ICONST 1.000000stack=[ 6.000000 1.000000 ]
0042: iadd      stack=[ 7.000000 ]
0043: store     8      stack=[ ]
0045: br       26      stack=[ ]
0026: load      8      stack=[ 7.000000 ]
0028: load     23      stack=[ 7.000000 7.000000 ]
0030: ilt       stack=[ 0.000000 ]
0031: brf      47      stack=[ ]

```

Функции

Шаблон для функции / процедуры :

```

($
(
    defun < имя >    (params < формальные параметры через пробел >
        (      [$]    < body >
            [ ( setResult! < индификатор , откуда
                в специальный регистр возвращаемого значения ,
                будет записано значение
                ) ]
            return
        )
    )
)
(
    defun < обязательно >::=main ( pass )
    ($ call < имя > ( args < фактические параметры > )
    // используем букву z – туда вернула функция значение
    )
)

```


)

Пример :

```

($ (// Пример с возвращением значения функции)
  (defun my (params a)
    ($(set! b (arif a + 100))
      // Записываем в специальный регистр,потом
      можем использовать его значение через буквы z)
    (setResult! b)
    (return)))
  (defun main(pass)
    ($)
    (// Аргумент как число)
    (call my (args 200))
    (// В букве z возвращенное значение последней функцией)
    (set! s (arif z + 300))(// ожидается 600)
    (print s)
    ))
  )

```

Вывод ВМ :

```

start_ip:17
0017: noop          stack=[ ]
0018: ICONST 200.000000stack=[ 200.000000 ]
0023: call    0,      1stack=[ ]
0000: load     0       stack=[ 200.000000 ]
0002: store    0       stack=[ ]
0004: load     0       stack=[ 200.000000 ]
0006: ICONST 100.000000stack=[ 200.000000 100.000000 ]
0011: iadd           stack=[ 300.000000 ]
0012: store     1       stack=[ ]
0014: store_re  1       stack=[ ]
0016: ret           stack=[ ]
0026: load_res      stack=[ 300.000000 ]
0027: ICONST 300.000000stack=[ 300.000000 300.000000 ]
0032: iadd           stack=[ 600.000000 ]

```

10 04.01.19 16:08:09

```
0033: store    18      stack=[ ]  
0035: print          print: 600.000000  
stack=[ ]
```

Проект на github : https://github.com/kosta2222/proj_Ukuvchi_Lang

Документация на компилятор и ВМ : <https://our-ruzaevka.herokuapp.com/rest/ddoc/index.html>