**ІНФОРМАТИКА**

*UDC* 004.42

# BASIC SEMANTICS OF COMPOUND PYTHON OPERATORS

**B. Hoshko, V. Chernyakhivskyy**

*Ivan Franko National University of Lviv,*
*Universytets'ka Str., 1, Lviv, 79000, Ukraine,*
*e-mail: volodymyr.chernyakhivskyy@lnu.edu.ua*

The semantic definitions of compound Python programming language operators, for which we have expanding applications in application projects, are explained in the article. The definitions are written in the basic initial form, which is primary for the study of programming and for the construction of algorithms for data processing. On the basis of semantic definitions, syntax definitions are formed, which form the correct subset of the general Python syntax, and can be supplemented by extension operations without modification of the base part.

The method of operational-functional definition of semantics is presented, which allows to rebuild syntax definitions so as to preserve the basic semantics of individual constructions of the language and to minimize the length of output by grammatical definitions *startdef=>+"example"*. For this purpose we use basically three methods: reduction of complete grammatical rules; substitution of definitions of non-terminals on the right side of the syntax; adding non-strictly defined terminals with reference to previously defined non-terminals.

Semantics models are built on the classification of operations and data conversion functions. Models are treated as universal algebras $U(A) =< M; \Omega >$, where M is a nonempty set (quantities, memory cells, structures, operators), and $\Omega$ is a set of operations (possibly partial) on the set M, including the signature. We define semantics models in two groups: 1) models based on valid Python operations; 2) models based on Python operators and control structures. For each group of models of algebra semantics has its own peculiarities.

Semantics models are defined for the following objects: numeric types, assignment, data system, print, input; conditional expressions, conditional operator if; loop operators while, for; try exception control operator; operator with context managers.

*Key words*: semantic definitions, Python, universal algebras, operations, data transformation functions, compound operators.

## 1. Introduction

The semantics of the programming language is determined by specifying the basic functions of data processing, a set of control structures and methods of constructing more "complex" programs based on "simple". The semantics of the programming language must be formally defined, otherwise it will not be possible in the future to build a corresponding speech processor. Today, there are two main areas for determining the semantics of programming languages: methods of denotational semantics and methods of operational semantics. Methods of denotational semantics are based on the corresponding algebras, methods of operational semantics are based on the syntactic structures of programs.

The syntax of the programming language defines a set of syntactic constructions of the programming language that are used to notation (record) the semantic units in the program.

## 2. The overview and task formulation

The means of defining the semantics of programming languages and formal analysis are an integral part of the general logic of mathematics and computer science [1]. The semantics of programming languages require different ways of determining them, depending on the applications of the language, and remain an urgent problem [2]. Denotation and operational semantics are a prerequisite for the construction of grammatical and semantic processors [3], [4] for both compiled-type and interpreted-language languages, to which Python belongs. Methods for determining and applying semantics are described, in particular, in [4], [5]. Our article is an attempt to build the primary semantics of compound Python operators for the purposes of learning programming and algorithm design, in view of the significant expansion of language use today.

The full definition of Python syntax [6] is presented as a combined list of Full Grammar specification parser rules, starting with the "single_input" start symbol (or another specified in [6]) and ending with the terminal characters of the language. The direct and complete application of the rules is ineffective for learning the language and for programming. The same applies to the syntactic definitions of individual constructs, for which non-terminal characters have links to other syntactic definitions, forming a large length of output $N$ in grammatical terms *startdef=>+"example"*.

The paper describes the method of operative-functional definition of semantics, which allows reconstructing syntactic definitions in such a way as to preserve the basic semantics of individual language constructs and maximally reduce the length of the output of N. For this we use mainly three techniques: the reduction of complete grammar rules; substitution of non-terminology definitions; the addition of terminals, defined unsteady, with reference to the previously studied material. A similar approach for the case of the C++ language is given in [7].

We construct semantic models based on the classification of operations and data transformation functions. Models are treated as universal algebras $U(A) =< M; \Omega >$, where M is a nonempty set (values, memory cells, structures, operators), and $\Omega$ is a set of operations (possibly partial) on the set M, including the signature.

Models of semantics are defined by two groups: 1) models based on permissible operations of the language Python; 2) models based on operators and Python control structures. For the first group of models, we construct the algebra of the form $U_A^i =< t_i; \{op_{ti}\} >$, where $t_i$ – the type of Python program object, $op_{ti}$ – defined operations for $t_i$. For the second group of models, we construct the algebra of the form $U_A^i =< \{par\}_{si}; \{operator_{si} : read, write\} >$, where $\{par\}_{si}$ – parameters of the implementation of the operator or the structure $si$, $\{operator_{si} : read, write\}$ – the function of converting input values or states of objects (values) read into output values or write states.

## 3. Semantics of compound operators and some elements of language

### 3.1. Numerical types, assignment, data system, print, input

These Python elements do not belong to compound operators. However, they are always needed in any program, following the usual steps: input data; data processing;

output results. In addition, such elements are required to organize the application testing procedure. We will also use them to outline the following article material.

$U_A^{ass} =< name, number, assignmentstatement; \{abs(), int(), float(),$
$math.sqrt(), math.sin(), +, -, *, /, //, \%, **\} >$
$U_A^{print} =< name, number, print\_function; \{print(), object\_list, sep, end\} >$
$U_A^{input} =< name, input\_function; \{input(), input\_line, convert\_to\_string, return\} >$

For such models we formulate syntactic definitions:

```
name ::= "first letter"  "the next letter" *
number ::=  "cipher"  "cipher" *
assignment_statement ::=   name  "="  arith_expr
arith_expr ::=  term   ( ( "+" | "-" )  term ) *
term ::=  factor   ( ( "*" | "/"’ | "%" | "//" )  factor ) *
factor ::=   name | number |  "built-in functions"  | power
power ::=  factor  "**"  factor
print_function ::=  print ( "object_list", sep=’ ’, end=’\n’ )
input_function ::=  input ( [ prompt ] )
```

In this definition, the complete list of permissible arithmetic operations was reduced, substitutions were made to the arithmetic expression arith_expr, added non-strictly defined "first letter terminals", "the next letter", "built-in functions", and "object_list".

Example of application:

```
any = int (input (’Print three digit integer:’))
hundred = any // 100; there = any // 10 % 10; one = any % 10
print (hundred, ten, one, sep = ’\ n’) # digits in the column
print (one * 100 + ten * 10 + hundred) # number in the reverse order
```

The first step would be prepare models and definitions as partial. The next step is to add objects, operations, and options to receive the full definition, while retaining the original model.

## 3.2. Conditional expressions, conditional operator if

$U_A^{if} =< \{or\_test, and\_test, not\_test, comparison, if, else, elif, suite\};$
$\{operator_{if} : if(True) \to suite1, if(False) \to (else\ suite2),$
$if(False) \to pass, if(False) \to (elif\ suite3)\} >$

Syntactic definitions:

```
comparison ::=  "arith_expr"  ( comp_operator  "arith_expr" ) *
comp_operator ::=  "<" | ">" | "==" | ">=" | "<=" | "!="
or_test ::=  and_test  ( "or" and_test ) *
and_test ::=  not_test  ( "and" not_test ) *
not_test ::=  "not" not_test | comparison
if_stmt ::=  "if" comparison ":" suite
             ( "elif" comparison ":" suite ) *
             [ "else" ":" suite ]
suite ::=   statement  NEWLINE  |  ( NEWLINE INDENT statement DEDENT ) +
statement ::=  "one_stmt" (  ";"  "one_stmt" ) *  [ ";" ]
```

Expressions in comparisons of "comparison" have been reduced only to the arithmetic "arith_expr" with reference to p.3.1, considering the non-strictly defined terminal. In comparison comp_operator did not include operations "in" and "is", we include them in the next step of the model. The verification of the operator condition was determined only as a comparison, except for extended conditions and lambda expressions. The set of suit operators was rebuilt so that all internal if statements are write with the offset to the right, or all in the same line with if. We define the "statement" with the help of a non-strictly defined terminal "one_stmt", considering the operator of any other type or the same kind as if.

Example of application – determining the number of the quarter of a plane:

```
if x>0 and y>0:
    quarter=1; print("x>0 and y>0", "quarter=1")
elif x>0 and y<0:
    quarter=4; print("x>0 and y<0", "quarter=4")
elif x<0 and y>0:
    quarter=2; print("x<0 and y>0", "quarter=2")
elif x<0 and y<0:
    quarter=3; print("x<0 and y<0", "quarter=3")
else:  print("x=0 or/and y=0")
```

### 3.3. Operators of the cycle while, for

$U_A^{while} =< \{conditional\_expression, or\_test, and\_test, not\_test, comparison, suite\}; \{operator_{while} : if(True) \to suite, if(False) \to (complete, pass), if(False) \to (else, pass), if(break) \to (complete, pass), if(continue) \to (skip, testing)\} >$

Syntactic definitions:

```
while_stmt  ::=  "while" conditional_expression ":"  suite
                [ "else" " : " suite ]
conditional_expression  ::=  comparison  |  or_test
```

Like the operator if, with reference to the "comparison", expanded conditions and lambda expressions were excluded in the first step. We specified the "conditional_expression" of a single conditionality request only for comparison operations "comparison" or only logical "or_test" operations without comparisons; in the general case, "or_test" stores the inclusion of "comparison" comparisons as in paragraph 3.2

An example is whether the number p is simple:

```
x = p // 2 # divisors to half the value of the number p
while x> 1:
    if  p % x == 0:  # divide remainder
print (p, 'has a divisor', x, 'is not simple')
break  # go through block
    else:
 x - = 1 # decrease by one
else: print (p, 'number is simple')
```

Semantic model for:

$U_A^{for} =< \{iterable\ object, parametr, in, iter(), each\_item, next(), order, suite, StopIteration\}; \{operator_{for} : if(iterator == next) \to suite,$

$if(break) \to (complete, pass), if(continue) \to (skip, testing),$
$if(iterator == StopIteration) \to (else, pass)\} >$

Syntactic definitions for a partial model of the first step:

```
for_stmt ::=  "for" exprlist "in" testlist ":" suite [ "else" ":" suite ]
exprlist ::=   NAME
testlist ::=   "compound_object" | NAME | "range(start, stop[, step])"
```

Non-terminal "exprlist" is defined as the first step only as a separate name NAME, which will be the operator of the loop. The non-terminal "testlist" was built differently – the non-strictly defined terminal "compound_object" was identified, separately we made reference to the generator of integer numbers range() as the primary form for the research of the statement "for".

Example and equivalent semantics:

```
for elem in [1,"two",(True,3)]:  print(elem,end="-")
#   1-two-(True, 3)-
it = iter( [1,"two",(True,3)] )
print(next(it),end="-");print(next(it),end="-");print(next(it),end="-");
print(next(it));  # most recent call last
#   1-two-(True, 3)- . . . most recent call last  . . . StopIteration
```

## 3.4. The "try" exception control operator

$U_A^{try} =< \{protected\ block, IndexError, ValueError, types\ of\ errors, probable\ problem,$
$try, except, finally, event\ registration, a\ spare\ algorithm, final\ operations\};$
$\{operator_{try}:$
$if(noexception) \to (skip\ except, goto(finally\ and\ continues\ after\ try\ statement)),$
$if(exception) \to (break\ protected\ block, search\ exception\ handler, goto\ except,$
$execute\ except\ block, goto\ finally, continues\ after\ try\ statement),$
$if(exception\ and\ no\ matches\ the\ exception) \to$
$(break\ protected\ block, goto\ finally, continues\ on\ the\ invocation\ stack)\} >$

Syntactic definitions for a partial model of the first step:

```
try_stmt ::=  "try" ":" suite
               ("except"  [ "the type of error" ]  ":"  suite)+
               ["finally" ":" suite]
```

The first step of the syntax definitions is not to include parts of the "try" operator, like "else" and "raise" statements. The optional "else" part can be functionally replaced by the "finally" part, and the "raise" statement can be viewed separately in the Simple statements section. In the exceptions part, the `(expression ["as" identifier])` definition of error type and method of processing exception handlers was replaced by the non-strictly defined terminal "the type of error".

Example: read a group of numbers from a text file, convert it to a numeric form, and save it for future reference:

```
# line of text file 'uniontry.txt':  "1  2  3  4  abc  6  7  8"
# error recording the fifth number
data = open('uniontry.txt','r')
try :
  group = data.readline().rstrip()
  xm = [ int(x) for x in group.split() ]   # list of numbers
except :             # for all errors
  xm = [ 0 ] * 10    # fix - we define 10 zeros
finally :
  data.close()       # final operation regardless of error
print(xm)            # result:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

## 3.5. Operator "with" context managers

$U_A^{with} =< \{context\ manager, context\ manager\ operator\ block,$
$try \ldots except \ldots finally\ usage\ patterns, \_\_enter\_\_()\ method, \_\_exit\_\_()\ method,$
$standard\ context\ manager's\};$
$\{operator_{with}:$
$obtain\ a\ context\ manager(expression) \rightarrow context\ manager's\ \_\_exit\_\_()\ is\ loaded$
$\rightarrow call\ \_\_enter\_\_()\ context\ manager's \rightarrow execute\ suite,$
$context\ manager's\ \_\_exit\_\_()\ method\ is\ invoked\} >$

Syntactic definitions:

```
with_stmt ::=  "with" item  ":"  suite
item ::=  expression "as" variable
```

To determine the basic semantics, we have the following syntax assumptions: 1) in the header "with" there can be only one element "item", the multiple elements are deferred to the next steps of the semantic definitions of operators; 2) we always associate the expression value of "expression" with some fixed "variable", considering it as an integer; We defer the variable as part of the whole object to the next steps; 3) we consider binding "expression" to a "variable" as obligatory.

Example. Scan text file, search for lines that contain the phrase "with operator":

```
with open("example.txt", 'r') as mf :  # file context manager
  whlines = [ aline.rstrip() if "with operator" in aline else None
                     for aline in mf ]
```

## 4. Conclusion

The operational-functional method can be used to build the definition of the semantics of other Python constructions. For each definition of semantics, we construct the corresponding syntactic definitions, performing the first step of reduction, substitution and addition of non-strictly defined terminals. This approach involves a certain order of the structure of definitions in order to minimize the entire semantic structure.

The basic semantics of compound operators, described in the article, are used for the construction of didactic materials for studying programming at the university, for the testing of application programs, as well as for the basis of algorithms for data processing.

## References

1. *Roşu G.* Matching logic / G. Roşu // Logical Methods in Computer Science. – 2017. – Vol. 13, No. 4. – P. 1–61.
2. *Chen X.* Matching mu-Logic / X. Chen, G. Rosu // Proceedings of theThirty-Fourth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). – 2019. – 24–27 June 2019. – Vancouver, Canada.

3. *Ahrent W.* Deductive software verification – the key book; from theory to practice / W. Ahrent, B. Beckert, R. Bubel, R. Hähnle, Ph. Schmitt, M. Ulbrich, editors // Lecture notes in computer science. – New-York: Springer, 2016. – 10001.
4. *Binsbergena L. T.* Executable component-based semantics / L. T. Binsbergena, P. D. Mosses, C. N. Sculthorped // J Logical Algebraic Methods Program. – 2019. – No. 103. – P. 184–212.
5. *Ştefănescu A.* Semantics-based program verifiers for all languages / A.Ştefănescu, D. Park, S. Yuwen, Y. Li, G. Roşu // Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16). ACM. – 2016. – Amsterdam, Netherlands 30/10-04/11/2016. – P. 74–91.
6. Python Software Foundation [US] // Python 3.7.3 documentation [Electronic resource]. Available from: https://docs.python.org/3/
7. *Kardash A. I.* Construction of basic algebraic operational-functional models for the study of programming of integer arithmetics / A. I. Kardash, V. V. Chernyakhivsky // Visnyk of Cherkasy State Technological University. Series of technical sciences. – Cherkasy: Editorial and publishing center of the ChTTU. – 2011. – No. 3. – P. 77–82.

*Article: received* 29.08.2019
*revised* 17.11.2019
*printing adoption* 20.11.2019

## БАЗОВА СЕМАНТИКА СКЛАДЕНИХ ОПЕРАТОРІВ МОВИ PYTHON

**Б. Гошко, В. Черняхівський**

*Львівський національний університет імені Івана Франка,*
*вул. Університеська, 1, Львів, 79000,*
*e-mail: volodymyr.chernyakhivskyy@lnu.edu.ua*

Викладено семантичні означення складених операторів мови програмування Python, для якої маємо розширення застосування в прикладних проектах. Семантичні означення записано в базовій початковій формі, яка є первинною для вивчення програмування і для будови алгоритмів процесів обробки даних.

На підставі семантичних означень складено синтаксичні означення, які утворюють правильну підмножину загального синтаксису мови Python, і можуть бути доповнені операціями розширення без модифікації базової частини.

Викладено метод операційно-функціонального визначення семантики, який дає змогу перебудувати синтаксичні визначення так, щоб зберегти базову семантику окремих конструкцій мови і максимально скоротити довжину виведення за граматичними означеннями *startdef=>+"example"*.

Для цього використовуємо головно три прийоми: скорочення повних граматичних правил; підстановка означень нетерміналів в праві частини синтаксичного визначення; додавання терміналів, визначених нестрого, з посиланням на нетермінали, визначені раніше.

Моделі семантики побудовані на підставі класифікації операцій і функцій перетворення даних. Моделі трактуємо як універсальні алгебри $U(A) =< M; \Omega >$, де M – деяка непуста множина (величин, комірок пам'яті, структур, операторів), а $\Omega$ – сукупність операцій (можливо частинних) на множині M, враховуючи сигнатуру.

Моделі семантики визначаємо двома групами: 1) моделі на підставі допустимих операцій мови Python; 2) моделі на підставі операторів і керуючих структур мови Python. Для кожної групи моделей алгебра семантики має свої особливості.

Моделі семантики визначені для таких об'єктів: числові типи, присвоєння, система даних, print, input; умовні вирази, умовний оператор if; оператори циклу while, for; оператор try контролю виняткових ситуацій; оператор with контекстних менеджерів.

*Ключові слова*: визначення семантики, Python, універсальні алгебри, операція, функція перетворення даних, складений оператор.