Python Semantics in K 5.0

Qianyang Peng, Wenhua Lin

December 20, 2018

Abstract

This is the report of the final project of CS522: Programming Language Semantics. We implemented a simple semantics for the Python Language, based on the Python Semantics implemented by Dwight Guth on K 3.2.1, and the untyped KOOL semantics for K 5.0.

1 Introduction

For the course project of CS522, we implemented a simple python semantics for K framework 5.0. The structure of the whole project is based on the *Python 3.3 Semantics* implemented by Dwight Guth on K 3.2.1. That is, we have a Makefile for compilation, a python script for source code preprocessing, and a bash file for script execution. For the implementation detail, including the K configuration, basic infrastructure and the implementation of some functionalities, our code is based on the *KOOL* project for K 5.0, which is a small object oriented programming language syntax implemented by K team.

For this course project we implemented the following components:

1. Python code parser;
2. Python arithmetic operator syntax and logical operator syntax;
3. Python variable declaration;
4. Python class and object instantiation;
5. Python for loop and range;
6. Python array, nested array and array multiplication;
7. Python print;
8. A test module that compares the output of our python framework with real python.

2 Python Semantics Overview

Python is an untyped object oriented programming language. Compared with a common untyped object oriented semantics in K framework, KOOL, python mainly has the following three language features that are more intelligent and flexible:

2.1 Variable Declaration

In KOOL, the syntax of variable declaration is like is JAVAScript. That is, we need to explicitly declare that a new identifier is a variable. For example:

```
var x;
x = 1;
```

For the code above, we can allocate the memory for variable x on stack with the first statement, and then assign the integer value *1* to x with the second statement.

Howeverm in python, we do not need the first statement. The only statement we need to do the same thing is:

```
x = 1
```

That is to say, python is able to intelligently detect whether a variable is already allocated in memory or not, and we do not need to explicitly declare a new variable.

2.2 Class Instantiation

In KOOL, the class instantiation operation is identified by the "new" keyword:

```
mary = new Student();
```

This distinguishes the instantiation of a class object from the function call. For example, if *Student*() is the name of a function and we want to assign the return value of this function to variable *mary*, the code will be:

```
mary = Student();
```

However in python, there is no difference in syntax that an instantiation of a class object and a function call. The python code for the two operations above will be the same, which is:

```
mary = Student()
```

That is to say, python is able to intelligently detect whether a call by name is to a class or to a function. In our implementation, we should be able to do it automatically.

2.3 Class Variables

In KOOL, every variable declare in a class is considered as member variable, however in python, there are different types variables in class. For example, for the code below:

```
class A:
a = "Static Variable"
def __init__(self):
    a = "Local Variable"
    self.a = "Member Variable"
    print(self.a, "is Member Variable")
    print(A.a, "is Static Variable")
    print(a, "is Local Variable")
print(A.a, "is initialized when declared")
test1 = A()
print(test1.a, "should print Member Variable")
```

In python, the output of the code above is:

```
Static Variable is initialized when declared
Member Variable is Member Variable
Static Variable is Static Variable
Local Variable is Local Variable
Member Variable should print Member Variable
```

Therefore, when implementing the python semantics in K, we need different initialization strategies for different variable types. For static variables, which is declared inside a class definition and outside any class method, the memory is allocated and the value is initialized when the class is declared, thus we store the static variables in a separate configuration tag. For member variables, which is declared inside a class method and declared as "self.x", is bonded to objects thus should be stored inside the object closure. For the local variables, which is declared inside a method but without a "self.", should be just stored inside the current environments.

2.4 Other Functionalities

Apart from the three main functionalities stated above, we also implemented some other pythonic functionalities, for example the arrays and nested arrays, array multiplication with integers, and some syntactic sugars that is common in python.

3 K Framework Syntax Overview

K framework is a rewrite based semantic framework. As there is no official documentation for the latest version of K 5.0, here we briefly introduce the syntax we used in our python semantic implementation.

3.1 Module

Using different modules can help grouping the language features. Usually we put the language syntax and semantics in separate modules.

We use the "require" and "imports" keyword to connect different modules. "require" keyword is followed by a .k file name which contains the language features needed for the definition. Usually, this is the file where the imported module is defined in. The file path is specified by the relative directory that is relative to the current file or "k/k-distribution/include/builtin".

The "imports" keyword is followed by a module name. After importing module B into module A, we can use the syntaxes, rules and configurations that are defined in module B when we are implementing module A.

K framework has the mechanics to prevent duplicated module import, so it is fine to transitively import a module more than once, without causing a redefine error.

3.2 Syntax

According to the definition of wikipedia, in computer science the syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language. In K 5.0, there are some default syntax definitions located in module DOMAINS-SYNTAX, including Id, String, Int and Bool.

3.2.1 Attributes

Attributes is usually assigned as the side condition of a syntax. Usually a syntax attribute is assigned to specify the associativity, the strictness and the order of evaluation,

left	: To tell the parser that the syntax production is left associative.
bracket	: To tell the parser that the parenthesis is associated to the program.
binder	: To tell the parser that "lambda" or "mu" binds to variables.
strict	: To tell the parser that to tell k the corresponding construct has a strict version strategy. That is, all arguments should be evaluated before the semantic rule is applied.
strict(i)	: This is selective strictness. To tell the parser that which argument of syntax production is strict.
seqstrict	: To tell the parser that the evaluation order is from left to right.
non-assoc	: To tell the parser that the syntax is non-associative.
function	: Used when the syntax declares a function definition.

3.2.2 Predefined constant identifiers

Predefined constant identifiers are variables starts with $. The program to execute cannot contain identifiers with these names and these names are specifically reserved for the semantics.

3.3 Rules

Rules are introduced by the keyword "rule". It is the main logic component of a language semantic and shows how the raw code could be reduced into the final result.

3.3.1 Variables

By convention, variables in the rule should start with a upper case letter. Variables could be tagged with the type associated to them. If tagged, a type checking at run time will be made. Tagging is formally saver but may make the executing speed slower.

3.3.2 Symbols

- "=>": Used in rules to separate the left hand side and right hand side of a rule.
- "∼>": This is a construct for K used in a computation structure, or computation. Read as "followed by" or "then".

3.3.3 Attributes

Similar to the syntax section, the rules also could have attributes as the side conditions.

macros	: To tell the parser that the rule is the interpretation of a syntactic sugar.
structural	: To tell the parser that a rule is desugaring but not a computational step. It is a light rule similar to "macros".
read	: To tell the parser that the rule reads value from the input buffer.
print	: To tell the parser that the rule prints out value to the output buffer.
lookup	: To tell the parser that the rule looks up the value of some specific key in a map.

3.3.4 Side conditions

Side conditions can only be boolean expressions in K. It is declared after a K rule using the "when" keyword. A boolean expression is usually constructed by bool operators such as "$= / = K$", "$== K$", "$>= K$", "$<= K$".

3.4 Configurations and Cells

Configuration are constructed with cells. It starts from a "configuration" keyword and being followed by some XML-style labels which indicates the start and return point of cells. It is very useful in our python semantic implementation because it makes it possible for us to implement more advanced language features such as input/output streams, static variables, variable declarations and object instantiations.

4 Configurations

In the configuration we can define some data structures as the runtime memory of our program. In our implementation, we mainly have 5 configuration areas:

1. $< env >$: This is the current local environment. It is a group of mappings from identifiers to the locations in $< store >$.
2. $< control >$: This is the memory when doing the control flow operations about the method closure and the object closure. For example, $< fstack >$ in it stores the environment when the program enters a function, and the memory state is restored from the function stack when the function is returned.

3. $< classes >$: This contains the definitions of and the inheritance relationship between classes.

4. $< storeClass >$: This contains a list the names of classes. We need it to distinguish a class identifier from a method identifier. As in python code class and method initialization is using the same syntax, we need this tag as a helper memory.

5. $< classStaticVars >$: This stores the static variables of classes. The memory here is initialized when the class is defined.

5 Project Structure

1. ./parse/parse.py: This script is the code parser that parses raw python code into the code that *krun* can read. We need this because we find it difficult to let the K framework to handle the indents in the program to generate proper block. Also, it is more convenient to handle the comment in the code in the preprocessing step than handling the comments in the syntax rules.

2. ./kpython: This is a bash script that takes the name of a python file as input, and output the printing results of our python semantics. What it does is just to parse the python code and invokes krun to execute the parsed code.

3. ./tests: This is a folder of test python scripts. When running "make test", all scripts in it will be executed by python3 and our python framework, and the output result will be stored in the ./tmp folder. If the results are the same, the test case will be marked as "PASSED", otherwise it will be marked as "FAILED".

4. kpython-*.k: These files are the modularized definition of our python semantic.

5. Makefile: This is the makefile, we hope users can use it to compile the code and run tests.

6 Conclusion

For the course project of CS522, we learned to use K framework 5.0 and practiced to implement a popular programming language semantic. K framework is a modularized, flexible and powerful framework, and we could implement almost every functionality we were planning to implement. With the help of proper configuration, we are able to implement more complicated environments and rules that defines more advanced programming language behaviors.