Python Semantics in K 5.0

Qianyang Peng, Wenhua Lin

December 19, 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 arithmic operator syntax and logical operator syntax;
3. Python variable declaration;
4. Python class and object instantiation;
5. Python for loop and range;
6. Python list and nested list;
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 KOOL, which is another untyped object oriented programming language implemented in K 5.0, python has some more intelligent and flexible language features. In our project, we mainly implemented the following language features:

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 stream, static variables, variable declaration and object instantiation.