Documentation: Simple Interpreter in Python

Philipp Kochanski, Joshua Schmidt

September 9, 2016

Abstract

In this project we implemented an extension to the Simple Interpreter written in Python. During the semester we build our own implementation of the object model and the interpreter based on the given parser and lexer. Therefore we used our implementation of the interpreter for this project.

1 Boolean

In order to write more complex programs with combined logic constraints for example within if statements we implemented the Boolean object extending W_NormalObject. This type has several built-in methods to combine boolean expressions like and, or, not, nor, nand, xor as well as xnor. The semantics are like `b1 and(b2)` where b2 can be another nested method returning a boolean value.

2 String

The String object extends W_NormalObject and is true if the string's value is not empty. This type provides some additional built-in methods to be able to combine or compare several strings. It is possible to concatenate two strings using `s1 append(s2)` or reverse a string with `s reverse`. Furthermore the length of a string is given by `s len`.

In order to compare two strings a call of `s1 equals(s2)` returns a W_Boolean value, i.e. `True` if the two strings are equal or `False` otherwise. The latter method can be used within boolean logic, e.g. within an if statement's condition to control the syntactic behaviour of the program.

3 Dictionary

The dictionary extends W_NormalObject and is internally implemented using a Python dictionary. Relating to the semantics of a dictionary we decided to represent each element as a tuple of key and value like it is realized in Python,

i.e. using the form `key:value` for its elements. We decided W_String, W_Integer as well as W_Float to be valid keys for a dictionary.

In order to work with dictionaries the object provides several built-in methods. We can add an element to a dictionary using `dict add(key,value)` or delete an element by its key with the use of `dict del(key)`. To derive the value of a stored tuple the object provides a getter that needs the key as the parameter like `dict get(key)`. The amount of the elements stored in the dictionary is obtained by `dict len`.

To be able to iterate over a dictionary's keys or elements we provide the method `get_keys` that returns a W_List of all keys. Furthermore we decided that the types of the keys are dynamic, i.e. we accept mixed types of keys in a single dictionary.

Despite that it is possible to check if a dictionary contains a key by calling `dict contains(key)` which returns a W_Boolean. Hence, we are able to build more complex conditions used in if statements which leads to more flexibility in the programs that can be processed by the interpreter.

3.1 Key-Value Tuple

Since we decided to implement dictionaries like it is done in Python, i.e. using the semantic `key:value` for the elements, we need to provide such an object. Essentially a W_KeyValue object is just a wrapper for two values semantically splited by a colon. Like described in [**??**] valid keys are of the type W_String, W_Integer or W_Float. This object does not provide any built-in methods because it is just used within dictionaries.

4 Conclusion

All in all we decided to implement the mentioned extensions to the interpreter to provide the in our opinion most important language features that have to be available before implementing other fancy extensions like a graphic interface or coroutines. The current state of the interpreter is quite mature and can be used to write reasonable programs using the Simple language.