Scolkam
Programming Language Proposal
COMS W4115 Programming Languages and Translators
Prof. Stephen Edwards
February 10, 2016

Steve K. Cheruiyot (skc2143)
Yekaterina Fomina (yf2222)
Connor P. Hailey (cph2117)
Léopold Mebazaa (lm3037)
Megan O'Neill (mo2638)

Summary

The Scolkam programming language is a subset of the Python programming language that compiles down to LLVM. Scolkam is therefore a general-purpose, kind of multi-paradigm[1] programming language. This language is mostly a technical exercise. However, Scolkam would also offer the opportunity to implement some features that are not existing natively, or poorly implemented, in Python. Multi-line comments, multi-threading, static typing for functions are among various suggestions obeying to that logic.

Implementing a subset of an existing programming language requires one to define the boundaries of that subset. We are not clear about what the perimeter of that subset should be. Therefore, we have divided this document in two sections. In the first section, we discuss what seem to be the minimal, bare-bones requirements of a Python-like language. In the second section, we mention different features that we could add on top of these minimal requirements. We also divided these additional features into four categories: the ones that add a new paradigm to Python (object-oriented or functional), the ones that are just more advanced Python features, the ones that are not well-implemented by Python, and the ones that could be very useful add-ons but seem too complicated to implement.

Minimal Language Features

Indentation

The syntax, like Python, would not define the beginning and the end of a function or a branch by using curly braces, but rather by indentation. We expect our parser to behave like the Python parser described in this document[2]. Namely, it would be able to detect different styles of indentation. For instance, indenting a block of code with 1 tab or 2 spaces or 4 spaces or 2 tabs is equivalent as long as the indentation is consistent. It would probably use some kind of stack to correctly parse the indentation. If this parser proves too cumbersome to implement, we may compel users to write *end* to show the end of code-block.

Comments

Inline comments would stay the same as in Python. We would like to add our own version of multi-line comments. In Python, they are so counter-intuitive to write that even the creator of the language deems it a "pro tip"[3]. We would enable users to write multi-line comments by putting them between two pairs of #. [4]

[1] In its smallest form, Scolkam is closer to an imperative programming language. Proposed extensions of the language can make it also object-oriented or functional.
[2] This is a link. URL: http://www.secnetix.de/olli/Python/block_indentation.hawk
[3] URL: https://twitter.com/gvanrossum/status/112670605505077248
[4] We are aware that there are parsing challenges with writing multi-line comments that close to inline comments. Therefore, we may very well change the syntax of these multi-line comments.

```
a = 0.0 # This is a comment
x = 3
## This is a multi-line comment
z = 3  ##
y = 3
# a, x, y are defined
# z is not defined
```

Data Types

As in Python, all data is represented by an object. These objects are immutable. (We are thinking of adding mutable data structures to the language —see Section "Potential add-ons".) We are not sure of the architecture of data storage, but we are not planning to make any distinction between some kind of "primitive" data types and objects.

- *NoneType*: A constant that would hold the value *None*, and that would be used when a item is undefined. The complete behavior of that type has yet to be defined. Notably, we still have to determine whether users are able to define objects with a undefined value.
- *Numeric types*: *int* for signed integer values, *bool*, to hold *True* or *False* values, and *float* for signed floating-point values. The length of these types would probably be 1 or 2 bytes. (Python's integers have unlimited precision[5], so this will be a departure from the original language.)
- *Sequence types*: *tuple* to store immutably a list of data objects.
- *Text sequence types*: *str* to store a list of characters. We are not quite sure yet whether we should allow for Unicode support or stick to ASCII.

Static Typing

In contrast to Python, our language would be statically typed. (For a discussion on dynamic typing, see the Section "Potential add-ons") There are multiple levels of dealing with static typing:

- The most constraining way would be to force the users, as in Java or in C, to declare the type of their variables next to them. This would give something like this: `int variable = 4`. To us, that seemed too constraining, and pretty remote from the notably relaxed Python syntax.
- Instead, we considered that, as long as users have to initiate their variables when they declare it, we could infer the type of the variable at declaration. (We may be wrong on that, but it seems to us that even strongly statically typed languages such as OCaml do not require the users to declare the type at declaration-time.) This would be equivalent to putting the equivalent of a C++ *auto* identifier in front of every variable declaration.
- More problematic is whether we should compel users to declare the types of the arguments and output of their functions. We have decided to do so here, (see subsection "Misc.") although we consider as a possible add-on to infer the types of these arguments and outputs. (See Section "Potential add-ons")

[5] URL: https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

Operations

- *Truth Value Testing*: *and*, *or*, *not*
- *Comparisons*:
  - == and ! =, to check the equivalence of two objects (same value),
  - *is not* and *is*, to check the equality of two objects (same memory address),
  - <, <=, >, >= to rank objects
- *Arithmetic*:
  - +, −, ∗, /, % — integer operations,
  - +., −., ∗., /., — floating-point operations

Control Flow

- Branches: *if*, *else if*, *else*
- Loops
  - *while* — Behave like one would expects, repeating the loop as long as the condition is still met
  - *for* — This loop would be implemented differently than in Python. Since we do not have native range data structures, if a user wants to loop over a range of number, the syntax will be `for i in 1..10` instead of `for i in range (1, 10)`. A user can also iterate on a tuple. (We discuss the possibility of making iterators for some objects in Section "Potential Add-ons".)

Misc.

Some other previously undiscussed aspects of the programming language basically copy Python.

- Function declarations follow the same model as Python, with the keyword *def*, with the exception that types are explicitly mentioned. The syntax of type declaration is following the model of Python Static Typing Checker mypy[6]: `def function-name([type: arg-name]-list) -> type`.
- New lines indicate the end of a statement.
- We would give users the ability to print strings, via a *print* statement (as in Python 2) or function (as in Python 3). In the following examples, we assume a Python 2-like statement.

Sample Examples

Summation

[6] URL: http://mypy-lang.org

```
def sum(int: x) -> int:
  if x == 1:
    return x
  else:
    return x + sum(x - 1)
```

GCD

```
def gcd(int: x, int: y) -> int:
  if y == 0:
     return x
  else:
     return gcd(y, x % y)
```

Hello, World

```
def hello() -> None:
  output = "hello, world!"
  print output
```

Iteration

```
numbers = (1,2,3) # Tuple definition
for number in numbers:
  print number
```

Potential add-ons

This section exists because we are unsure about the feasibility of certain features. Instead of producing a half-backed, unrealistic proposal, we have made the choice to list potential paths for us to enrich the language. We have tried to assess as much as we could the utility and the feasibility of these different features. We will, however, need the instructors' help to decide which paths, if any, to follow. In order to clarify this section, we have separated the different ideas into broad categories.

Adding a paradigm to Python

Class and object management

Python has a significant object management portion. It would therefore be useful to implement some parts of it. Here are multiple existing Python features that could be implemented. These features are ranked in the order of difficulty, and, in general, except for inheritance, we assume that a feature cannot be implemented without having the features listed directly above in this list.

- *Attributes*: The ability to create a class with attributes. (For instance, a 3D vector would have attributes *x*, *y*, *z*.) A thing that Python enables but that would apparently be complicated to do in LLVM is to add attributes on the fly, meaning outside of the scope of the class definition. Therefore, we may want to skip that.

- *Initializer*: The ability for users to write *__init__* functions. A more complicated question would be whether we could implement both normal initializers (*__init__()*) and copy initializers (*__init__(other-object)*).

- *Object Functions*: The ability for users to write functions that are attached to an instance. More complicated would be to implement some common function decorators:

  - *@classmethod* (for shared methods that have the instance as the first argument),
  - *@staticmethod* (for methods that do not have the class or the instance as first argument),
  - *@property* (for read-only methods)
  - *@[attr-name].setter* (for methods that set the value of a specific attribute)
  - *@[attr-name].deleter* (for methods that delete a specific attribute, which would be)

  This would be indeed nice perks to have, but we are worried about the level of entropy that it would incur.

- *Inheritance*: Inheritance in Python mainly exists by subclassing classes (put another way, extends it). We know that inheritance is a major hurdle to implement in LLVM, so we might not be sure that it is necessarily a good idea to implement.

- *User defined default functions*: The ability for users, as in C++, to modify some functions that are common to each members, such as the addition operator (*__add__*), the conversion to boolean (*__bool__*), iterator functions, etc. That would require us to not only define the functions for default types, but also taking into account cases in which these functions are not defined for some classes. This would probably be too much of a difficulty.

Functional programming

Python enables users to do some kind of functional programming. Functions are first-class objects in Python, so we could think about enabling that. Multiple functional programming languages have been written in LLVM, however we do not yet know how feasible this is.

Disparate advanced features

- *Other (mutable) types*: *complex*, *list*, *range*, *dict*. We have got to confess not to grasp fully the implications of mutability in LLVM. That being said, it seems clear that it is, according to the LLVM documentation itself, a very fastidious[7] feature to implement.

- *Other control flow operations*: *break*, *continue*. This does not seem very complicated to implement, however we are weary of overstuffing the language.

- *Casting*. This would probably be very useful, although it may require to implement bitwise operations (in Python: >>, <<, |, &, *compl*). It would also enable us to implement arithmetic operations that are closer to vanilla Python. For instance, the / operator could output integer or floating-point results depending on the arguments.

[7] URL: http://llvm.org/docs/tutorial/LangImpl7.html

- *String formatting.* We were thinking of something under the format:
  `z = "This string is comprised of %s and %s" % (x,y)`
  or something closer to concatenation:
  `z = "This string is comprised of " + x + " and " + y.`
  In the later case, just the implementation of the add operation to strings could be sufficient.

Features that do not exist in Python

- *Type detection across the language.* We have said earlier that the minimal language we could do would be strongly statically typed, and would detect the type of an object at declaration time. However, the types of argument and output data of functions would have to be declared. We could think of removing this declaration by automatically detecting the types of these arguments and outputs. This has been done by multiple languages, including OCaml, and even though the Hindley-Minler type system could be of great use here, none of us has sufficient experience to implement that right away.

- *Multi-threading* — Although there are Python modules for multi-threading[8], they are, from what one of the TAs told us, pretty bad and inefficient. LLVM apparently implements multi-threading pretty easily, and therefore adding good multi-threading could be a good addition to the language.

Ideas that seem far-fetched to us

These ideas seem very difficult to implement. Since we know that we may underestimate their feasibility, we still wish to mention them here.

- *An equivalent to* ***`raw_input`***. The idea, here, is to have some kind of *prompt* function that would get user input. This would have significant dynamism to the language. However, there are multiple challenges to this. One, we had the impression, after browsing the Web to find ways to express user prompt in LLVM, that the implementation of that feature is fairly system-dependent. Two, one enormous challenge of user input is input security, and, even though our language would have a fairly small scope, we probably would not have time to spend crafting even basic input security features such as type detection.

- *Dynamic typing.* Python is dynamically typed, and this provides a great level of comfort to the user, who does not have to think about the types of the variables it declares. The issue, here, is LLVM basically allows programmers to implement dynamic-typed languages only if they are ready to die[9]. It seemed way out of the scope of the class.

[8] URL: http://www.tutorialspoint.com/python/python_multithreading.htm
[9] URL: http://stackoverflow.com/questions/6833068/why-is-llvm-considered-unsuitable-for-implementing-a-jit