SPY - Project Proposal

Simplified **Py**thon

A strongly typed, general purpose language inspired by Python

Sanil Shah (ss4924)

Description

SPY is a strongly typed, easy to use language inspired by features from Python and functional languages like OCaml and SML. It is further inspired by the work done in the JSJS project from Spring 2016 (JSJS). The goal is to create a type-safe version language with functional elements that can be used easily by Python programmers with a similar syntax that will compile down to Python. A lot of functional programming languages (like LISP) can often have complicated syntax that is hard to understand and unfamiliar to most programmers. Since Python is quickly becoming language of choice for introductory programming classes, SPY aims at offering a familiar yet functional alternative to first-time Python users and programmers who wish to dip their toes into functional programming. The language will not have all the robust features of Python, but will contain a small subset that can be used to create fairly robust algorithms

Features

SPY will be **strongly typed.** While Python is a dynamically typed language, SPY will offer a type-safe alternative, making programs less prone to bugs that can be easily caught by the compiler. Types will be inferred by the compiler and will not have to be explicitly specified.

All variables will be **immutable.** Again this is different from how Python deals with variables and more similar to OCaml but he goal will be to offer a similar syntax as Python without the same freedom to write hacky code. All assignments will be treated as values that cannot be changed. This implies that values once they are given a name, cannot be reassigned to a different type in the existing scope.

Similar to OCaml, **everything will be an expression** in SPY. This is a key concept in most functional languages and will remove the need for an explicit `return` statement. SPY will include a `void` type (similar to `unit` in OCaml) for expressions which have side effects like printing.

SPY will have **no iteration.** The `for` loop and `while` loop syntax from Python will be removed in favor of functional constructs like `fold`, `map` and recursion. This will ensure that the language is used to write functional code instead of iterative code that most Python users might be used to.

SPY will allow **functions as first class members** similar to both Python and OCaml. Functions can be passed to other functions as arguments or returned from functions. SPY will also support anonymous functions (without a name) similar to Python or OCaml.

SPY will provide libraries for **complex types** such as `List`, `Dict`, and `Tuple`. Since all types in SPY are immutable these built in types will provide immutable implementations of basic data structures that are used available in Python. Since SPY is a strongly typed language, unlike Python, all elements in the `List` must have the same type. Similarly a `Dict` may only contain keys of the same type as well as values which may be of a different type than keys but must be of the same type as all other values in the `Dict`. The `Tuple` type will allow an arbitrary number of elements of different types but once again it will be immutable.

Lexical Convention

Basic Conventions

Keywords	`true, false`
	`and, or, not`
	`def, lambda, if, else, elif`
	`cons, len, hd, tl, filter, map, fold, append`
	`set, get, keys`
	`print`

Comments	`# this is a comment`

Conditionals	`if x < 5:`
	`  "small"`
	`elif x < 100:`
	`  "big"`
	`else:`
	`  "huge"`

Primitive Types

The will not be explicitly specified in the code, however they will be inferred by the compiler. The compiler will raise an error if the rules for type safety are not met.

Data Type	**Example**
bool	`true, false`
num	`10, 10.0, 0.0001, -10`
string	`"Hello", "world"`
unit	`print "Hello world"`

Operators

Operator Name	Syntax	Applicable Types
Assignment	`a=5, a=true, a=[], a={}, a=(4,5), a="hi"`	`num, string, bool, list, dict, tuple`
Arithmetic	`x+y, x-y, x/y, x*y, x%y`	`num`
Equal / Not equal	`x == y, x != y`	`num, string, bool, list, dict, tuple`
Comparison	`x>y, x<y, x<=y, x>=y`	`num, string`
Logical	`x and y, x or y, not x`	`bool`
Concat	`x ^ y`	`string`

Lists

```
# Creating a list with num values
a = [1, 2, 3, 4]

# Creating a list with string values
b = ["a", "b", "c", "d"]

# Creating a list with mixed types (compiler error)
c = [1, "a", true, "b"]

# Getting the length of a list (returns 4)
d = len(a)

# Getting the head of a list (returns 1)
e = hd(a)

# Getting the tail of a list (returns [2, 3, 4])
f = tl(a)

# Add to the front of a list (returns [0, 2, 3, 4])
g = cons(0, f)
```

Dictionaries

```
# Creating a dict with string keys and num values
a = {"cat": 1, "dog": 2}

# Creating a dict with string keys and list values
b = {"cat": [1, 2], "dog": [3, 4]}

# Creating a dict with mixed value types (compiler error)
c = {"cat": [1, 2], "dog": 3}

# Creating a dict with mixed key types (compiler error)
d = {"cat": "animal", 3: "dog"}

# Getting the keys of a dict (returns ["cat", "dog"])
e = keys(a)

# Getting the value associated with a key (returns 1)
f = get(a, "cat")

# Setting the value for a key in a dict
# (returns a new dict {"cat": 3, "dog": 2})
g = set(a, "cat", 3)
```

Tuples

This data type can be used to easily group multiple values similar to tuples in Python or OCaml. This allows functions to return multiple values easily.

```
# Creating a tuple with 2 values of type (string * num)
a = ("cat", 1)

# Accessing elements of the tuple (returns "cat")
b = get(a, 0)

# Setting elements in a tuple (returns a new tuple ("dog", 1))
c = set(a, 0, "dog")
```

Functions

Function blocks are delimited using tab spacing (similar to Python). Further since there are no return statements and everything is an expression the first line in a block that isn't assigned to anything, a print statement or not an if (elif or else) condition is automatically returned. Any lines following that line will result in a compiler error or warning indicating that the code is unreachable.

```
# Defines a function to add two numbers
def add(x, y): x + y

# Defines an anonymous function to add two numbers
add = lambda x, y: x + y

# Defines a function to filter even numbers from a list
def getEvent(numbers):
    checkEven = lambda x: x % 2 == 0
    filter(checkEven, numbers)
```

Sample Code

```
# Euclid's GCD
def gcd(a, b):
    if a == b:
        a
    elif a > b:
        gcd(a - b, b)
    else:
        gcd(b - a, a)

# Combines two lists into a list of tuples
def combine(x, y):
    if len(x) != len(y) or len(x) == 0:
        []
    else:
        cons((hd(x), hd(y)), combine(tl(x), tl(y)))

# Calling combine with incorrect arguments (compiler error)
a = combine(1, 2)

# Calling combine correctly (returns [(1,3), (2,4)])
b = combine([1, 2], [3, 4])
```