SPY

Language Reference Manual

Sanil Shah | ss4924

Introduction

SPY is a strongly typed, easy to use language inspired by features from Python and functional languages like OCaml and SML. 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 C. 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. The language design aims at emphasizing type safety while minimizing the number of keywords, keeping the syntax as similar to Python as possible. The language will provide library functions to perform common tasks instead of reserving keywords.

Python vs. SPY

SPY is inspired by Python and the syntax is as close to Python syntax as possible. SPY is type safe but the types will be inferred by the compiler so the language will not need special keywords to specify the types. In fact, like Python there will be no explicit way to specify types, however type safety will be ensured by the compiler once types are inferred.

Comments

Python	SPY

`# This is a single line comment`
 	
`"""`
`This is a`
`multi line comment`
`"""`
	`# This is a single line comment`
	
	`Multiple line comments aren't allowed.`
	`Anything after a # symbol on a line will`
	`be ignored and treated as a comment.`

Literals

Python	SPY

`42`
`0.42`
`True`
`False`
	`42`
	`0.42`
	`true`
	`false`

Strings

Python	SPY
`'Hello world'`	`Single quoted strings aren't allowed`
`"Hello world"`	`"Hello world"`
`"Hello \n world"`	`"Hello \n world" (escape special chars)`
`"Hello" + "world"`	`"Hello" ^ "world"`
`"Hello" + 5`	`Mixed type concat isn't allowed`

Lists

Python	SPY
`[1, "hi", 0.42]`	`Mixed type lists are aren't allowed`
`[1, 2, 3]`	`[1, 2, 3]`
`[1] + [2, 3]`	`1 :: [2, 3] (cons)`
`[1, 2] + [3, 4]`	`[1, 2] @ [3, 4] (append)`

Dictionaries

Python	SPY
`{"foo": 1, "bar": "baz"}`	`Mixed value dictionaries aren't allowed`
`{"foo": 1, "bar": 2}`	`{"foo": 1, "bar": 2}`

Functions

Blocks and scoping in Python are denoted by white-space (either tabs or an equal number of space characters). This often leads to confusion and results in hard to find errors in syntax since white space is invisible. Instead, SPY will use the "end" keyword to denote the end of a block, and white-space will be ignored just like any other programming language. This will apply to conditional statements as well as function definitions. New lines are used to determine the end of an expression. All other white-space will be used purely for convenience and readability.

Python	SPY

`def add(x, y):`
`  return x + y`
 	
`add = lambda x, y: x + y`
	`def add(x, y):`
	`  x + y`
	`end`
	
	`add = lambda(x, y): x + y`

Conditionals

Python	SPY

`if x < y:`
`  return 42`
 	
`if x < y:`
`  return 42`
`elif x > y:`
`  return -42`
`else:`
`  return 0`
	`This isn't allowed in SPY. The else construct is required.`
	
	`if x < y:`
	`  42`
	`elif x > y:`
	`  -42`
	`else:`
	`  0`
	`end`

Types

Types are dynamically inferred at runtime in Python and will similarly be inferred in SPY as well so that they do not have to be explicitly specified. The inferred types are described here to make compiler messages more understandable and in case explicit typing is necessary at any time in the future. The words used to describe the types will be reserved as keywords.

Primitive Types

There are five primitive types in SPY.

1. **void:** This is a special type used when expressions do not return a value. This is the case for statements like print or file reading operations that may be supported. These expressions have side effects (such as printing to stdout) but do not return a value and hence have a type void.
2. **bool:** This represents a logical value which can either be true or false. The keywords "true" and "false" are reserved to represent literals of this type.
3. **int:** The int type is used to represent literals between +2^31 and -2^31. Literals with no decimal points or floating parts will be inferred as integers.
4. **float:** The float type is used to represent numbers containing a decimal part or an exponential portion. Numbers like 0.42 or 1e5 will be inferred as floating point numbers. Floats will be stored in a double in C as mentioned in the Python numeric type guide here.
5. **string:** The string type is used to represent literals containing text. It is a sequence of characters encoded in the ASCII format. Each character occupies 16 bits and will follow in order from the starting position which will contain the first character of the string. Strings will be inferred as any sequence of ASCII characters between double quotes or based on the usage of the caret operator.

Functions to allow casting between various types will be provided by the standard library.

Complex Types

There are two complex types in SPY.

1. **list:** The list data structure can be used to store a sequence of values. This can be useful when keeping track of multiple primitives that need to be referred to repeatedly. It serves as a way to express one or more related values together. Unlike Python, all elements of a list is SPY must contain the same type of element. Lists can also contain other lists so long as every element list contains the same type of elements.
2. **dict:** The dictionary data structure is another important feature of SPY and are similar to dictionaries in Python. However unlike Python, all keys in the dictionary must be of the same type as well as all the values. Keys and values may have different types but all the keys have to be homogenous and all the values have to be homogenous. The type of the dictionary will be inferred during initialization or when it is accessed.

Function Types

Like everything else in SPY, functions are expressions as well and evaluate to expressions. The type of a function is represented by a list of types of its formal arguments and its return type which may be one of the primitive or composite types, or a function type in itself.

Polymorphic Types

Types that cannot automatically be inferred reliably or unambiguously will be treated as polymorphic or generic types. For example a map function can be applied to a list containing elements of any types. Hence the type of the map function is polymorphic in that it can accept arguments that are integer lists or string lists or even lists of lists. SPY allows for polymorphic types for such functions and lambda functions as well that may apply to more than a single type of formal arguments.

Type Declarations

Explicit type declarations aren't permitted in SPY (just like Python). However assignment works as follows and enables type inference.

```
a = 1 # int
b = 1.5 # float
c = -0.00005 # float
d = "cat" # string
e = true # bool
f = [1, 2, 3, 4] # list(int)
g = ["a", "b", "c", "d"] # list(string)
h = {"cat": 1, "dog": 2} # dict(string, int)
i = {"cat": [1], "dog": [2]} # dict(string, list(int))
```

Lexical Conventions

The rules followed by the parser and the lexer to break the program into acceptable tokens are listed below. The rules for syntax are very similar to Python with minor differences but are listed out in completeness below to remove any ambiguity.

Comments

Only single line comments are allowed in SPY. Anything on a line following a "#" is considered a comment and ignored by the compiled. Multi line comments are not supported in SPY.

```
# This entire line is a comment
a = "cat"   # The rest of this line is a comment
```

Lines

Each line in SPY represents a separate expression. We will talk about expressions later in this manual. Since there is no way to delimit expressions (like a semicolon in JS or Java), each line will be treated as a separate expression. SPY will recognize the linefeed (ASCII: '\n') as a line termination sequence.

Whitespace

All non line termination white-space characters can be used to separate tokens. This includes horizontal tab ('\t'), form feed ('\f') and carriage return ('\r). The white-space characters will be used exclusively to separate tokens and hold no other meaning.

Indentation

Unlike Python, SPY doesn't follow a strict indentation policy to represent code blocks. Instead the end of a code block will be indicated using the "end" keyword provided in the language. This allows users to use any indentation they want. The "end" keyword will be used to demarcate function definitions as well as if-else blocks.

Identifiers

Identifiers are used to represent various entities in SPY and to store their values. All reserved SPY keywords mentioned in the section below or types mentioned above cannot be valid identifiers. Further, reserved C keywords will not be valid identifiers either. Identifiers can be named starting with at least a single lowercase letter, followed by any number of lowercase letters, uppercase letters, digits or underscores. Identifiers will not start with an uppercase letter, a digit or an underscore.

Identifier regular expression:

```
id = ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

Valid Identifiers	Invalid Identifiers
x	X (cannot start with uppercase letters)
name	1stName (cannot start with a digit)
angle1	name&age (invalid character)
last_name	lambda (reserved SPY keyword)
dateOfBirth	int (reserved SPY type)

Keywords

The below keywords are reserved and have special meaning in SPY. The are reserved for use by the programming language itself and may not be used as names of identifiers elsewhere in the program. They serve a special function and cannot be used to serve any other purpose. Keywords used by SPY are listed below:

```
def, lambda, end, print, if, elif, else, hd, tl, keys, and, or, not, true, false
```

Additionally reserved keyword types: `void, bool, int, float, string, list, dict`

Separators

This is the list of characters that separate tokens other than white-space as mentioned above. These are simple single characters that can be used to separate tokens and often have a special meaning when used in specified format to represent lists or dictionary literals.

Character	Separator
'('	{ LPAREN }
')'	{ RPAREN }
'['	{ LSQUARE }
']'	{ RSQUARE }
'{'	{ LBRACE }
'}'	{ RBRACE }
','	{ COMMA }
'.'	{ DOT }
':'	{ COLON }

Literals

Literals are used to represent values of primitive or composite types in SPY. There are six kinds of literals as listed below.

1. **Boolean literals:** The bool type literals can be two values, "true" or "false" represented by the ASCII characters that form the words. These are reserved keywords so they may not be used as identifiers and are written without the surrounding quotes to signify the boolean literals.

```
bool = true | false
```

2. **Integer literals:** Integer literals represent values of the int type. These can be a sequence of one or more digits starting with a non zero digit or a single zero. It can be used to represent numbers between 0 and 2^31. Negative integer literals will be created using the unary operator for negation which will be covered later in this manual.

```
int = (['1'-'9']['0'-'9']* | '0')
```

3. **Float literals:** Float literals represent values of the float type. They can be used to represent fractional or decimal numbers. Float literals have whole number part followed by a dot followed by a fractional part. The whole number part can be defined by the same rules as integer literals above. The dot is mandatory. The fractional part can be defined as a sequence of any number of digits.

```
float = (['1'-'9']['0'-'9']* | '0') '.' ['0'-'9']*
```

4. **String literals:** String literals are used to represent textual data. They are denoted by a sequence of zero or more ASCII symbols enclosed in double quotes. Certain special characters are denoted by escape sequences, which are a backslash followed by the special character. The following escape sequences are supported to account for special characters like tabs, new lines, and quotes: horizontal tab ("\t"), newline ("\n"), backslash ("\\"), double-quotes ("\"") and carriage return ("\r").

```
string = '"' ([^'"' '\'] | '\\' ['"' '\\' 'n' 'r' 't'])* '"'
```

5. **List literals:** List literals are used to represent values of the list type. They are denoted by a sequence of zero or more comma separated expressions surrounded by square braces. The square braces are required to specify a list literal. They may contain no elements in which case the list is considered to be empty. Examples of valid list literals are provided above under list.

6. **Dictionary literals:** A dictionary literal is used to store pairs of related values. This is similar to a map in other languages. It is denoted by a left brace ("{") followed by a list of zero or more comma separated key-value pairs and ending with the right brace. ("}"). The braces are required to specify a dictionary literal. The key-value pairs are denoted by an expression followed by a colon (":") followed by another expression. The first expression represents the key while the second expression represents the value. All keys must be of the same type and all values in the dictionary but also be of the same type, keeping the dictionary consistent.

Operators

The following characters are used as operators in the language and carry a special meaning that is described below. Their behavior is defined as part of the language and cannot be changed.

Character	Operator
'+'	{ PLUS }
'-'	{ MINUS }
'*'	{ MULTIPLY }
'/'	{ DIVIDE }
'%'	{ MODULO }
'^'	{ CONCAT }
'::'	{ CONS }
'@'	{ APPEND }
'<'	{ LT }
'<='	{ LEQ }
'>'	{ GT }
'>='	{ GEQ }
'='	{ ASSIGN }
'=='	{ EQUALS }
'!='	{ NEQ }
'<-'	{ SET }
and	{ AND }
or	{ OR }
not	{ NOT }
keys	{ KEYS }

Operators in SPY may be unary or binary. There are only two unary operators (logical negations "not" and arithmetic negation "-"). All other operators are binary and the types of operands that they may act upon is defined below. Binary operators are used in the infix notation and are written between two expressions that act as the operands.

Arithmetic Operations

Arithmetic operations require operands of either type float or type int. The five arithmetic operations are plus, minus, multiply, divide and modulo. The minus sign can also be used as a unary operator to negate either float or int numbers. The operands must be expressions that evaluate to literals of either float or int types. Both operands must be of the same type. The result of these operations is the same type as the input operands.

1. **Plus:** Add two values

```
1 + 2  # evaluates to 3
```

2. **Minus:** Subtract the second value from the first

```
3.0 - 1.5  # evaluates to 1.5
```

3. **Multiply:** Product of two values

```
5 * 2  # evaluates to 10
```

4. **Divide:** Divide the first value by the second

```
5.0 / 2.0  # evaluates to 2.5
```

5. **Modulo:** The remainder obtained when dividing the first value by the second

```
6 % 4  # evaluates to 2
```

6. **Unary negation:** The arithmetic negation of the given value

```
-3 # evaluates to -3
```

String Operations

There is only a single string operation. The caret ("^") symbol is used to concatenate two strings. Both operands must be string literals or expressions that evaluate to string literals. The result of the concatenation operation is also a string literal.

1. **Concat:** Combine the two strings into a single string

```
"cat" ^ "cat" # evaluates to "catcat"
```

Relational Operations

Relational operations require operands of either type float, int or string. The four relational operations are greater than, greater than or equal to, less than and less than or equal to. The operands are expressions that evaluate to literals of either float, int or string types or variables specified by identifiers of any of those types. Both operands must be of the same type. For string types comparison is done based on the ordering of the alphabet. The result type of these operations is of type bool (it may be either true or false).

1. **Less than:** Is the first value strictly less than the second value

```
1 < 2  # evaluates to true
```

2. **Less than equals:** Is the first value less than or equal to the second value

```
3.0 <= 1.5  # evaluates to false
```

3. **Greater than:** Is the first value strictly greater than the second value

```
"cat" > "cat"  # evaluates to false
```

4. **Greater than equals:** Is the first value greater than or equal to the second value

```
"cat" >= "cat"  # evaluates to true
```

Comparison Operations

There are two types of comparison operations. These are equals ("==") and not equals ("!="). These can be used with operands of any primitive or composite type. Once again both operands must be of the same type. The result of these operations is also a boolean. Comparison for composite types is done by structure (similar to python). This means that two lists will be equal if they contain the same elements in the same order, and two dictionaries will be equal if they contain the same keys with the same value for each key.

1. **Equals:** Is the first value logically, mathematically or structurally equal to the second value

```
"cat" == "cat"  # evaluates to true
"cat" == "dog"  # evaluates to false
3.0 == 3.0  # evaluates to true
3 == 4  # evaluates to false
[1, 2, 3] == [1, 2, 3] # evaluates to true
{1: "cat"} == {1: "dog"} # evaluates to false
```

2. **Not equals:** Are the two values logically, mathematically or structurally not equal

```
"cat" != "cat"  # evaluates to false
"cat" != "dog"  # evaluates to true
3.0 != 3.0  # evaluates to false
3 != 4  # evaluates to true
[1, 2, 3] != [1, 2, 3] # evaluates to false
{1: "cat"} != {1: "dog"} # evaluates to true
```

Assignment Operations

The assignment operator ("=") is used to assign a value to an identifier. The value may be of any primary, composite or function type. The identifier is on the left side of the "=" sign. The expression on the right side is evaluated and then assigned to the identifier on the left side.

1. **Assignment:** Assign the value on the right to the identifier on the left

```
a = 3 + 4  # a is assigned 7
b = "cat"  # b is assigned "cat"
```

Logical Operations

There are three types of logical operations. These are logical and ("and"), logical or ("or") and logical negation ("not"). These can be only be used with operands that are of the bool type. Any expression used with these operators must evaluate to a bool literal. The resulting type of these operations is also a bool type. The logical or and logical and operators are binary while the logical negation operator is unary.

1. **Logical and:** Are both the operands true. If not then this is false.

```
true and false  # evaluates to false
true and true  # evaluates to true
```

2. **Logical or:** Are either of the operands true. If not then this is false.

```
true or false  # evaluates to true
false or false  # evaluates to false
```

3. **Logical negation:** If the operand is true, this is false, if the operand is false, this is true.

```
not true  # evaluates to false
not false  # evaluates to true
```

List Operations

There are two types of list operations. These are cons ("::") and append ("@"). Both are binary operators used with the infix notations. The result of both operations is a list type. The correct usage for each is described below.

1. **Cons:** Adds the value on the left to the list on the right. The operand on the left must be of the same type as the elements in the list. The operand on the right must be a list.

```
3 :: [2, 1]  # evaluates to [3, 2, 1]
"cat" :: ["dog", "cow"]  # evaluates to ["cat", "dog", "cow"]
```

2. **Append:** Appends the two lists together. Both lists must have the same type of elements. Both operands must be valid lists.

```
[3] @ [2, 1]  # evaluates to [3, 2, 1]
```

```
["dog", "cow"] @ ["cat"]  # evaluates to ["dog", "cow", "cat"]
```

Dictionary Operations

There are three types of dictionary operations. This includes setting a value in a dictionary, getting a value from the dictionary and getting all the keys in the dictionary. Dictionary operations are different from all other operations in that they do not have a specific special character operator associated with them. However they do have a special syntax to access elements in the dictionary and to set values for the dictionary. Further the list of all the keys of the dictionary can be obtained by using the "keys" keyword. The usage for these is shown below.

1. **Get:** Gets the value of key in the dictionary. The return type of this operation is the same as the type of the values of the dictionary being accessed. The syntax is similar to Python. The name of the dictionary followed by the value of the key surrounded by square braces as shown below.

```
a = {"foo": 1, "bar": 2}
a["foo"]  # evaluates to 1
a["bar"]  # evaluates to 2
```

2. **Set:** Sets the value of a key in the dictionary. The value being assigned must be the same type as all other values in the dictionary. Further the key must be the same type as other keys already in the dictionary. The syntax is shown below using a "<-" operator.

```
a = {"foo": 1, "bar": 2}
a["baz"] <- 3 # a is now {"foo": 1, "bar": 2, "baz": 3}
```

3. **Keys:** The return type of this operation is a list of the same type as the keys of the dictionary. It can be used as a way to iterate over the values in the dictionary. Keys acts as a unary operator where the operand must be a dictionary and the result is a list.

```
a = {"foo": 1, "bar": 2}
keys a  # evaluates to ["foo", "bar"]
```

Other useful dictionary operations such as "find", "contains", "remove" or "clear" will be provided by the standard library.

Operator Precedence

The operator precedence specifies the order in which operations occur when there is more than one operator in an expression. All operators in SPY are left associative except for assign ("=") and cons ("::") which is left associative. The operator precedence can be overruled by using parentheses to dictate which operations occur first.

```
unary negation
*, /, %
+, -
<=, >=, <, >, ==, !=
not
and, or
^, ::, @, keys
=, <-
```

Functions

Functions are an integral part of any functional language. All functions in SPY are expressions. Functions are treated as first class citizens and can be passed to and from other functions as arguments or return values. Functions can be assigned to named identifiers using the "def" keyword as demonstrated above. Further smaller functions that can fit on a single line can be created using the lambda keyword with the syntax shown above. The argument list for functions is specified as a comma separated list of expressions containing one or more element and surrounded by parentheses. The parentheses are mandatory even if there are no arguments to the function.

Function declarations

Functions in SPY can be declared in two ways. Large functions with names can be declared using the within a "def" and "end" block. Single line smaller functions can be created using the "lambda" keyword syntax as shown below.

```
def func_name(arg1, arg2,...):
  # block of expressions
  # last expression is returned
end

lambda(arg1, arg2,...): # a single expression that is returned
```

This syntax can be parsed with the following structure in the grammar:

```
DEF ID LPAREN formals_opt RPAREN COLON block END    { FuncLit($4, Block($7)) }

LAMBDA LPAREN formals_opt RPAREN COLON expr         { FuncLit($3, Block([$6])) }
```

Above, formals_opts is a comma separated list of identifiers, block is a list of expressions separated by newlines in the code, and expr is a single expression that is returned.

The types of a function including the return type and types of its argument list will be inferred by the compiler. This represents the type of the function and the types of the arguments when calling a function must match the types that are inferred by the compiler.

Function invocation

A function is invoked by using its name which is declared during function declaration. A function must be invoked with actual arguments that match the types of the formal arguments for the function. These arguments can be any expression so long as the expression evaluates to the expected type for the formal argument. A function invocation evaluates to a value that is of the type which is the return type of the function being invoked. The syntax for function invocation is described below:

```
a = func_name(arg1, arg2,...)

# Examples
b = factorial(3)  # b is assigned 3 * 2 * 1 which is 6
c = isPalindrome("dog" ^ "cow")  # c is assigned false. dogcow isn't a palindrome
```

This syntax can be parsed with the following structure in the grammar:

```
ID LPAREN actuals_opt RPAREN    { Call($1, $3) }
```

Expressions

Everything in SPY is an expression. This allows an entire program to be defined as a list of expressions. Expressions are composed of identifiers, literals, operators, function definitions and function calls. Expressions can be of the following types:

1. Unary operations
2. Binary operations
3. Assignment operations
4. Function definitions
5. Function invocations
6. Literals of primitive types
7. Literals of composite types
8. Block of expressions (expression list)
9. Identifiers
10. If statement, optional elif statements and else statement

A program is a sequence of expressions separated by newline characters.

Blocks and Scope

A block is simply a list of expressions that are collectively treated as one. In SPY blocks are encountered inside function declarations and if-elif-else statements.

The syntax for if-elif-else statements is exactly like Python (without the tab spacing) and ending with the "end" keyword. All expression between the colon (":") after the if statement and the next elif or else are treated as a block. Similarly everything between the colon after the elif statement and the next elif or else is treated a block.

Identifiers in SPY are statically scoped so that they can only be accessed within the block in which they are declared. Global variables may exist and may be reassigned but they must remain of the same type throughout the program. If an identifier cannot be found in its current scope then the parent scope is checked recursively until we get to the global scope. If it is still not found then an error is returned if an unfound identifier is accessed.

Library Functions

List functions: List.length, List.filter, List.map, List.fold, List.reverse, List.contains
Dictionary functions: Dictionary.contains, Dictionary.remove, Dictionary.clear
IO functions: IO.print, IO.println, IO.input
Cast functions: Cast.int_to_float, Cast.float_to_int, Cast.string_to_int, Cast.int_to_string, Cast.string_to_float, Cast.float_to_string