Language description

This manual describes the syntax and (some) semantics of a simple procedural language. Please note that some of the examples in this description might be legal syntactically, but are not semantically. You should be mindful of those cases.

Lexical Description

Keywords Lexemes

- `if`
- `elif`
- `else`
- `while`
- `return`
- `and`
- `or`
- `not`
- `pass`
- `def`
- `bool`
- `int`
- `string`

Operator Lexemes

We support the following operators, which follow the operator precedence table from the language C:

- `==`
- `>`
- `>=`
- `<`
- `<=`
- `!=`
- `+`
- `-`
- `*`
- `/`
- `**`

Literal Lexemes

- bool: `"True"` or `"False"`
- integers (int): An integer literal can be a decimal or hex.

  Examples:

  ```
  100     : Decimal (cannot start with Zero if it is NOT zero)
  ```

- float (similar to float in C language)

  Examples: 3.14, 1.0, …

- string: A string is an array of characters. It is written as a sequence of characters enclosed by double quotes or single quotes. The closing quotes must be on the same line as the opening quotes. That is, they cannot be separated by a newline. To make things easier, a string cannot contain a double or single quote character, since this would terminate the string.

  Examples:

  ```
  "this is a string"        : simple string that contains 16
                                    characters
  "this is \"invalid\""     : invalid string, double quotes cannot
                                    be escaped
  "this is no newline\n"    : string that contains 20 characters,
                             including a backslash and a lowercase n
  ""                        : empty strings are okay
  ```

  <u>`Every example is also correct with single quotes ' instead of double ".`</u>

- identifier: An identifier can be a variable or function name. Identifiers must start with an alpha character (upper or lowercase letter) or single or double underscore(_ or __), followed by zero or more digits, "_", and/or other alpha characters, <u>it cannot start with a digit or more than 2 underscores.</u>

Other Lexemes

Lexem	Use	Example
`;`	Each statement ends with a semicolon	`i = 0;`
`,`	Used in variables and parameter lists	`x, y, z;`
`{`	Start block of code	
`}`	End block of code	
`(`	Begin parameter list	
`)`	End parameter list	
`[`	Begin string (character array) index	
`]`	End string (character array) index	
`=`	Assignment operator	
`->`	Return type	

Description of Program Structure

Programs

A program is composed of many functions, just listed one after another. Every legal program should have one and only one function: '__main__()'. This is case sensitive, so Main() is incorrect. Of course, a program can have user defined functions too. Any function must be defined before the point of call.

Correct (legal):

```
def foo()->int: {
      return 0;
}


def __main__(): {
        int a = foo();
}
```

Incorrect (illegal):

```
def __main__()->int: {
        int a = foo();  #foo is used before it's declared
}

def foo(): {
      return 0;
}
```

Functions

Functions are declared as:

```
def id '(' parameter_list ')' '->' type ':' '{' body '}'
def id '(' parameter_list ')' '->' type ':' statement ';'
def id '(' parameter_list ')' ':' '{' body '}'
def id '(' parameter_list ')' ':' statement ';'
```

Note the placement of the "()" , "{}" and ':' symbols. These must go exactly there, functions can be with a body enclosed with "{}" or a single statement, also return type is optional, in which case the function should not return a value(like void in C/C++).

- *def* – keyword def
- *id* – is an identifier and the name of the function.
- *parameter_list* – are the parameters you have declared for the function.
- *type* – return type of the function, optional.
- *body* – contains the body of the function, which are statement.

Nested functions are <u>not</u> possible with this language.

Correct (legal):

```
def foo(int x; float y; string z)->int:
{
        x = 5;
        y = 5+10;
        return 0;
}


def goo(int i, j; float k):
{
    foo(2, 3.5, "hi");
}

def hoo(bool a, b; int c)->int:
        return c;
```

Incorrect (illegal):

```
def foo():
      a = 15;
      return 5;         # only single statement allowed
```

Parameter List

A *parameter_list:* you can pass as many variables as you want, the list is separated into type sub-list with a semi-colon, where each sub-list starts with a type definition, followed by a comma separated list of variable names, you can also have variables with default value assigned, this default value must be a **literal**. However, all variables with a default value must be listed at the end of the type sub-list.

Notice that the default values make the function callable with different variations in length of parameter list.

Correct (legal):

```
def foo(int x, y:17; float z:15):
{
        return 0;
}
```

Incorrect (illegal): all default values should be at the end

```
def foo(int a:10, b; float c:3.147):      # parameter with a default
                                          value followed by one without
      a = 15;

def foo(int x, y: 7+14): { pass }      # default values can't be
                                       complex, only literals
```

Variable Declarations

Variable declarations should appear before variable usage, we can't declare midway, e.g. var can only be used after declaration of var itself, although we can declare and not use the variable.
They are to be used in the following way:

```
"type" ID1  ","  ID2  ","  ID3  ","  ...  ","  IDN  ";"
```

Variables can be assigned in the declaration.

Correct (legal):

```
int i;
float m = 3.14, n = 9.8, x;
```

Incorrect (illegal):

```
x = 10;                # if x wasn't declared beforehand

def foo(): {
        x=15;          # cannot be used before declaration
        int x;
}
```

Strings (character arrays)

Arrays are declared with the following syntax:

```
STRING ID1 "=" STRING_LITERAL ";"
```

Strings operations are allowed with the usage "[]", there're 3 ways to used this operator of a string: id[_], id[_:_], id[_:_:_], where the blanks are expressions, which can also be epsilon(as used in python), all 3 operations return a string.

Correct (legal):

```
string b = “moshe”;
string c = b[5];
string d = b[1+1:3];
string e = b[:5];
```

Statements

Statements can be many things: an **assignment** statement, a **function call** statement, an **if** statement, an **if-else** statement, a **while** statement, a **code block**, etc.

Body

A **code block** starts with a "{" and ends with a "}". It may contain variable declarations and statements. Both variable declarations and statements are optional. But, a code block cannot be empty, for a placeholder we can use **pass**. Of course, since a code block is a statement, code blocks can be nested within code blocks.

Correct (legal):

```
def foo(int i, j, k)-> int: {
        int total;                     # variable declaration
        total = square(k*(i+j));       # statements
        return total-k*j;
}

def foo():
{
    {
        { pass ;}               # pass code blocks are okay, although not
                                very useful, usually meant for
                                placeholders to be fulfilled later on
    }
    return 0;
}
```

Incorrect (illegal):

```
def foo():
{
    {    }                      # illegal in our language
    return 0;
}
```

Assignment

The syntax for an assignment statement is:

```
lhs "=" expression ";"
lhs1, lhs2 "=" expression1, expression2 ";"
```

Here, lhs -- which stands for left-hand side (of the assignment) -- specifies all legal targets of assignments. Specifically, our grammar accepts two different lhs items:

```
   x = expr;                 /* lhs is variable identifier */
   str[expr] = expr;         /* lhs is string element */
```

We also allow multiple assignment:

```
        x, y = 1, 2;          /* x, y are assigned with 1, 2
                               respectively */
```

We cannot assign values to arbitrary expressions. After all, what sense would make a statement such as:

```
        (5+2) = x;
```

Thus, we have to limit the possible elements that can appear on the left-hand side of the assignment as discussed above.

The right-hand side of assignments is less restrictive. It includes expressions, as well as string literals.

Function Call

The syntax for a function call statement is:

```
lhs "=" func_id "(" expr0 "," expr1 "," ... exprN ")" ";"

function_id "(" expr0 "," expr1 "," ... exprN ")" ";"
```

Correct (legal):

```
a = foo(i, j);
foo(i, j);
foo(i);         # in case j had a default value of declaration
```

If

An if statement starts with an **if** keyword with the syntax presented next, it can be followed by elif statement/statements with their respective expressions (conditions), and followed by else statement(there could be only one of these). elif/else could never appear without a if statement, these are also optional.

```
"if" expression ":" "{" body "}"

"if" expression ":" statement;

"if" expression ":" "{" body "}"
"else" ":" "{" body "}"

"if" expression ":" statement; "else" ":" statement;

"if" expression ":" "{" body "}"
"elif" expression ":" "{" body "}"

"if" expression ":" "{" body "}"
"elif" expression ":" "{" body "}"
"else" ":" "{" body "}
```

While

```
"while" expression ":" "{" body "}"

"while" expression ":" statement;
```

if/else/loop statements

Correct (legal):

```
if 3 > 2 :
{
    #...statements...
    i = 5;
}

if True : { j = 3; } else : { k = 4; }
while True: { l = 2; k = l + j; }
if (a>b and i==a+b) i = 5;     # this is also correct, since an
                               expression can be enclosed in brackets */
if x<=y : { j = 3; } else: x = x-1;
while False: x = x + 1;
```

Return Statement

As in python, return statement isn't mandatory, but when used, it should be the last statement of its block.

Correct (legal):

```
def foo()->int:{ return 0; }
def foo_2()->int: { int a = 2; return a+9; }
def foo_3()->int: { if (True): { return foo(); } return 0; }
def foo_4() { int a = 2; }
```

Incorrect (illegal):

```
def foo_3()->bool { return True; a = 5; }     # statement after return
```

Expressions

An expression's syntax is as follows:

```
expression operator expression
   OR
operator expression
```

Operators have the same precedence as in Python (Similar to C).

Correct (legal):

```
3 or 2
(3 + 2) / 3 - 5 * 2
True and False or False
5
3.234
True
-5
not x
a == b
a is b
```

Comments

Comments in this language are block comments (Python-style-like). The form is, only 1-line comments are allowed:

 # comments

Correct (legal):

```
# this is my comment
```

Incorrect (illegal):

```
// wrong language
/* wrong language */
# multiple lines
       Not allowed
```