The ENSEA language

ENSEA/FAME Computer Science

Language description

The purpose of this homework is to develop a programming language, called the *ENSEA language*, and write an interpreter for this language in Python. In the ENSEA language, everything is an expression. There exist three types of expressions:

1. Numbers: which can be either floating point numbers or integers. For example `1`, `1.34`, `-5`, etc.
2. Symbols: identified by a string of characters, like `x`, `var`, `If`, etc.
3. Compound expressions: of the form `s[arg1,arg2,...,argn]`, where `s` is a symbol and the *arguments* `arg1`, ..., `argn` form a non empty list of ENSEA expressions.

In particular, ENSEA expressions can be seen as trees, as shows Figure 1. This language is big enough

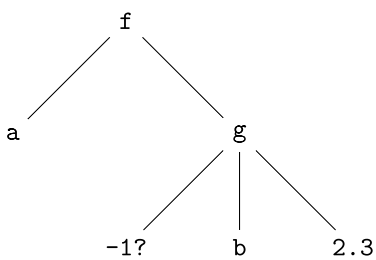

Figure 1: A tree representation of the expression `f[a,g[-1,b,2.3]]`

to define the factorial function in a recursive fashion. It needs less than 150 lines of Python code to write such an interpreter.

```python
symbol_table = {}

class Expression:
    def __init__(self, tag, args):
        self.tag = tag
        self.args = args

    def evaluate(self):
        #TODO to be modified
        return self

def parse(s):
    raise NotImplementedError

while True:
    expr = parse(input('ENSEA>>> '))
    print(expr.evaluate())
```

ENSEA expressions are implemented with the class `Expression` defined at line 3. The `tag` field contains the value of a node (which can be of type `int`, `float` or `str`) and the `args` field is either the empty list (if `self` is a number or a symbol) or a non empty list of arguments (if `self` is a compound expression).

For example, here is how to define the expression of Figure 1:

```
>>> minusone = Expression(-1, [])
>>> b = Expression('b', [])
>>> twodotthree = Expression(2.3, [])
>>> right = Expression('g', [minusone, b, twodotthree])
>>> a = Expression('a', [])
>>> expr = Expression('f', [a, right])
```

The paradigm is that each expression is *evaluated* by the interpreter. The `evaluate` method (at line 8) returns an *evaluation*. For the moment (line 10), this method returns an unmodified expression, yet we hope that soon enough the evaluation of `Plus[1,3]` be `4` (Question 3).

The global variable `symbol_table` is a dictionary that stores all the definitions associated to symbols. For example, we may want to assign an expression to a symbol, with instructions such as `Set[x,1]` and retrieve the "value" of `x` by looking in the table `symbol_table`. This issue will be raised in Question 8.

Finally, lines 15–17 constitute the top-level interactive prompt. It is an endless loop, where the user is asked to enter a string, which is then parsed into an expression, evaluated and printed.

Expression-String conversions

Question 1. In the class `Expression`, write a method `def __repr__(self)` returning a string representation of the ENSEA expression `self`.

Question 2. Write the `parse` function (line 13).

Hint: define a function `parse_comma` which parses a comma separated list of ENSEA expressions. For example `parse_comma('a,g[-1,b]')` should return the list

```
[Expression('a', []), Expression('g', [Expression(-1, []), Expression('b', [])])]
```

The functions `parse` and `parse_comma` are *mutually* recursive.

The evaluation procedure

We will gradually modify the `evaluate` method. We want the following behaviour when we evaluate a compound expression:

1. first we evaluate all its arguments,
2. then, depending on the tag of the expression, we launch a special procedure.

In step 2, if the tag is equal to `'Plus'`, we add all the evaluated arguments together, if all of those are numbers. If some of the arguments are not numbers, we return the expression unmodified. Figure 2 illustrates the two-step evaluation process.

Question 3. Modify the `evaluate` method so that it takes into account what has just been said.

You can dynamically check the type of a variable with the built-in function `isinstance`. For example, `isinstance(self.tag, str)` returns `True` whenever `self.tag` is a string.

Test with the following inputs:

```
ENSEA>>> Plus[1,Plus[2,3]]
6
ENSEA>>> Plus[1,2,x]
Plus[1,2,x]
ENSEA>>> Plus[x,Plus[3,1]]
Plus[x,4]
```

```
ENSEA>>> f[Plus[1,2],g[Plus[3,4]]]
f[3,g[7]]
ENSEA>>> Plus[4.5]
4.5
```

Question 4. The ENSEA language also contains a built-in symbol `Times` which accounts for multiplication. Modify the evaluator so that we can perform multiplication in the ENSEA language.

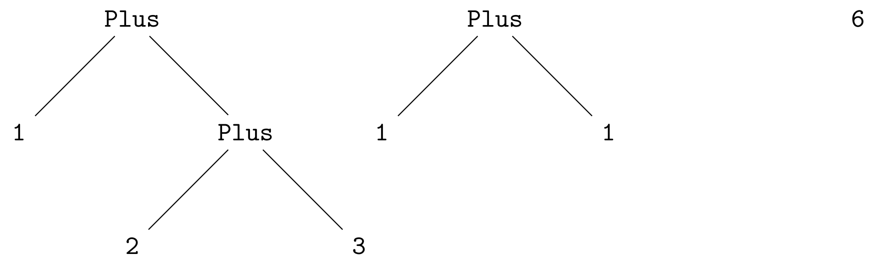

Figure 2: Evaluating `Plus[1,Plus[2,3]]`

Question 5. In the class `Expression`, write a method `def __eq__(self, other)` which returns `True` if `self` and `other` are identical expressions, and returns `False` otherwise. This method overrides `==`, so you may now use `==` (resp. `!=`) to test equality (resp. non equality) of expressions.

Question 6. The ENSEA language contains a built-in symbol `Equals` which tests equality of the arguments, *i.e* `Equals[arg1,arg2]` evaluates to the symbol `True` if `arg1` and `arg2` are identical expressions, and `False` otherwise. Do not confuse the Python Boolean value `True` with the ENSEA symbol `True` ! Test the following:

```
ENSEA>>> Equals[Times[2,3],Plus[1,5]]
True
ENSEA>>> Equals[2,a]
False
ENSEA>>> Equals[1,2,3]
Equals[1,2,3]
```

Non standard evaluation

So far, all arguments of a compound expression were evaluated before the whole expression was. However, there are exceptional cases where this is not desirable. In those cases, the evaluation procedure is *non standard*.

We begin with the `If` symbol. When we evaluate an expression of the form `If[arg1,arg2,arg3]`,

1. We first evaluate *only* the first argument.
2. If the evaluation of the first argument yields `True`, we return the evaluation of the second argument, otherwise we return the evaluation of the third argument.

Question 7. Incorporate the `If` symbol into the evaluator.

Now, we will move on to the `Set` symbol, which allows in the ENSEA language to make assignments. `Set[arg1,arg2]` also has a non standard evaluation, only its second argument is evaluated. Then, if `arg1` is a symbol, we store `arg2` in `symbol_table` at index `arg1.tag`. Finally, the evaluator returns the second argument.

For example, when evaluating `Set[x,3]`, the instruction `symbol_table['x'] = Expression(3, [])` is executed, and eventually the evaluation is `3`.

Question 8. Incorporate the `Set` symbol into the evaluator.

Question 9. Up to now, we only investigated how to evaluate compound expressions. Now, if `x` is a symbol, there two possibilities:

1. `x` is a *pure* symbol (no definition is attached to it),
2. `x` was previously assigned an expression.

In case `x` is pure, the evaluation of `x` returns `x`. Otherwise, it returns an evaluation of the expression assigned to `x`.

Note: to check whether `symbol_table['x']` exists, you can use `symbol_table.get('x')` which returns `None` in case `x` is a pure symbol, and `symbol_table['x']` otherwise.

Test the following lines:

```
ENSEA>>> Set[x,Plus[2,y]]
Plus[2,y]
ENSEA>>> Set[y,5]
5
ENSEA>>> x
7
```

Functions

The ENSEA language contains yet another non standard symbol `Function`. When we evaluate a compound expression with tag `Function`, none of the arguments are evaluated. Intuitively, the expression `Function[x,Times[x,x]]` is used to represent the abstract square function which maps x to x^2.

Question 10. Incorporate `Function` into the evaluator and test the following lines:

```
ENSEA>>> Function[x,Times[x,x]]
Function[x,Times[x,x]]
ENSEA>>> Function[x,Times[Plus[1,1],x]]
Function[x,Times[Plus[1,1],x]]
```

Question 11. In the class `Expression`, write a new method `def substitute(self, expr1, expr2)` which returns the expression obtained from `self` by substituting all occurrences of `expr1` by `expr2`. See Figure 3 for an illustration.

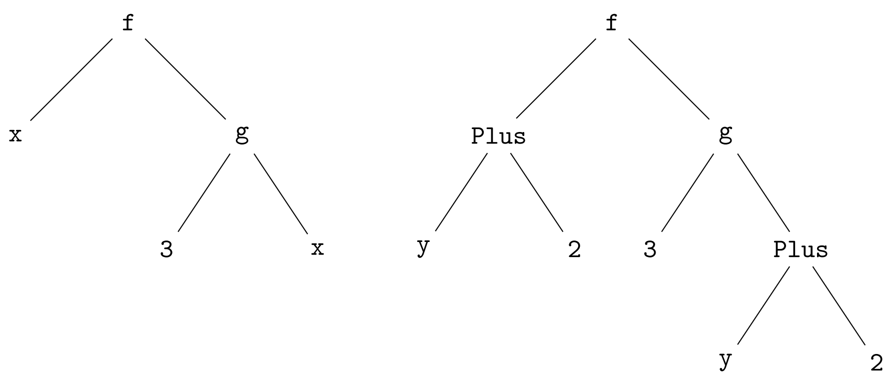

Figure 3: Substituting `x` by `Plus[y,2]` in `f[x,g[3,x]]`

We now want to apply a function at a given value. The way to do that in the ENSEA language is to use the `Apply` built-in symbol. See the following:

```
ENSEA>>> Set[f,Function[x,Times[x,x]]]
Function[x,Times[x,x]]
ENSEA>>> Apply[f,5]
25
```

Question 12. Add the `Apply` functionality in the evaluator. According to your own judgment, you have to decide whether `Apply` has a standard evaluation procedure or not.

Testing the language

You can now play with the ENSEA language. For example, here is how to define the factorial function:

```
ENSEA>>> Set[fact,Function[n,If[Equals[n,0],1,Times[n,Apply[fact,Plus[n,-1]]]]]]
ENSEA>>> Apply[fact,10]
3628800
```

Question 13. Create a function in the ENSEA language that computes the n-th term of the Fibonacci sequence.