CISS445: Programming Languages
Project

The goal is to write a python2 interpreter using OCaml.

Most of the previous assignments are useful. You want to go over them thoroughly.

Note that you are not translating to assembly. You are translating to OCaml. OCaml is powerful. So you want to make full use of that.

You do have some knowledge of grammars by now. You can use some of that knowledge. You can choose not to. You can also choose to use your knowledge of grammar parsing only in some parts of your code.

You can write your parser which builds parse tree differently. It's up to you. You don't have to be super efficient. In fact it's probably better if you can design a new parsing algorithm that is not very efficient, rather than use the what's in my notes. That way you open a new line of research. Note that even if you do not want to use the parser in the notes, you should still study them carefully. The grammar parsing ideas in the notes is not meant to be copy-and-paste material. It's to help you design your own parser. Have the grammar rules in front of you also help you think about the recursion needed to build the parse tree and hence help you think about the recursion on the computaton during runtime.

Main thing: get your interpreter working.

You are welcome to discuss with each other.

The basic features are:

- Python can be used as a calculator. The values your python interpreter should be able to handle includes integers, floats, and boolean values. Python support mixed mode arithmetic expression and automatic typecasting among these values. In other words you can have `1 + 3.5 + True`. Because Python is dynamically typed, it should be clear that you will need an OCaml type for all such python values.
- Your interpreter should implement all numeric operators and all boolean operators. You do not need to implement bitwise operators on numeric values.
- *** OPTIONAL *** You can handle string type. You do not need to handle any string operations.
- *** OPTIONAL *** You can try to handle python lists. Remember python lists are non-homogeneous. But if you can handle the above mixed mode arithmetic expressions, then this is pretty straight-forward. Note that this means you need to have an OCaml

type to represent Python lists. You have three options: you can use OCaml lists or your can use OCaml arrays or your own list type. I don't care how you do it as long as your interpreter works. There are many methods/functions for Python lists. You only need to implement `len`, `append`, bracket operator, and `in`. (Python's bracket operator notation allows slicing. This is optional.)

- At this point, if you have been lexing in an ad hoc way, you should really use ocamllex.
- Your interpreter can handle variables. You should use a symbol table or environment. This is just a collection of (identifier, value) pairs. Remember that the OCaml environment is a list. The standard Python interpreter actually uses a hashtable. Which data structure you use is entirely up to you. You will (obviously) need two functions, `lookup` and `update`, for your symbol table.
  - `(lookup symtable id)`: returns the value of identifier `id`
  - `(update symtable id val)`: return a new symbol table with `(id, val)` added.

  Your variables can appear in expressions. Your variables can also appear on the left-hand-side of the assignment operator. You need NOT implement the augmented operators (`+=`, `-=`, etc.)
- Your interpreter allows you to print all expressions. Since expressions evaluate to values, of course your print function prints all types of values supported. In general, it's a good to have a function to convert different values to strings first.
- At this point, I very strongly suggest you have the concept of statement. You can handle this through grammar and parsing, i.e., by building a tree structure for your statement (or rather different types of statements). If you choose not to do that, you can handle statements. in an ad hoc way – a statement can be just a list of tokens.
  Regardless of how you implement it, you should have a type for statement. (In particular, you want to have a concept of assignment statement.)
  A statement can affect the symbol table. An expression creates a value. For instance `x = y + z` is a statement. It changes the symbol table. But the expression `y + z` does not. Therefore if you have a function to execute a statement, that function should take in a table too. Therefore I suggest you have the following two functions:
  - `exec_stmt(stmt, symtable)` which returns a `symtable`.
  - `eval_expr(expr, symtable)` which returns a value.
- You might not realize this until later: You really need a concept of block of statements very early. You can use a list of statements as your block. Or you can construct a parse tree of statements.
- Your interpreter should allow you to write if statements. You want to try if statements of the form

  ```
  if a == 0: x = 1
  ```

  I very strongly suggest you consider an if statement a statement. Assuming you have statement type, then an if In other words, it's a good idea to have a constructor for the if statement that helps with building the parse tree. Also, for program execution, besides `exec_stmt`, you should have `exec_if_stmt`.

- Next, you want to try

  ```
  if a == 0:
      x = 1
  ```

  Here comes the issue of indentation. You want to compare with

  ```
  x = 0
  y = 0
  if a == 0:
     x = 1
     y = 1
  z = x
  ```

  Of course indentation is just a matter of counting left trailing whitespaces. How are you going to handle counting left trailing whitespaces while still using ocamllex? Note that in python, all blocks (other than the global) follows a statement that ends with a semicolon.
- Your interpreter should allow you to write if-else statements. (Hint: Write an OCaml function to handle if-else computation to make life easier.) You can forget about `elif`.
- Then you want to try *nested* if statements.

  ```
  x = 0
  y = 0
  if a == 0:
     x = 1
     if b == 0:
        x = 2
        if c == 0:
           x = 3
     y = 1
  z = x
  ```

- Now try nested if-else statements.
- *** OPTIONAL *** [REQUIRED AFTER SPRING 2022] Now try to handle while-loops. Here's an example:

  ```
  i = 0
  s = 0
  while i < 10:
      s = s + i
      i = i + 1
  ```

  It's clear how you should detect a while-loop: Check for the while token. To capture the boolean guard of the while-loop, after detecting the while token, in the same statement, detect the colon token. You should also have the detection of block of statements done before the parsing while loops. Also, I suggest not worrying about having indented blocks in the block of statements in your while loop – worry about one level of indentation first. You might want to have a function `exec_while_stmt(stmt, symtable)` which returns a symtable. Note that in python variables created in a while-loop statement is not destroyed, i.e.,

```
x = 0
while x < 3:
    y = 42
    x = x + 1
print(y) # <--- y is still around!
```

This is the same for an if statement and for-loop statement. The exception is the function block – local variables created in a function block are destroyed on return.

- *** OPTIONAL *** Your interpreter should allow you to write for-loops. Note that you have to finish list processing first.

  ```
  for i in [1,2,3]:
      print(i)
  print(i)
  ```

  What's the point of this warning? (Hint: Write an OCaml function to handle for-loop computation to make life easier.)
- *** OPTIONAL *** Functions are pretty tough. Don't.
- *** OPTIONAL *** Functions as first class value are pretty tough too. Don't.
- Error report can be minimalistic.