
COSMIX TO
PYTHON
COMPILER

Introduction

We are glad to present you computer compiler that translates one programming language into another and runs it. This compiler was developed within Formal Languages and Automata Theory class in Suleiman Demirel University (Qazakhstan, Qaskelen) by Young&Wild team members: Duisenova Alua, Kushenchirekova Dina, Zhandos Akbota, Dyussupova Alina and Berdibek Magzhan under responsive supervision of Birzhan Moldagaliyev in 2019 in terms of final exam. In this course were covered main principles of Automata and Languages in terms of computer machine.

The compiler makes a translation of an invented imaginary language called Cosmix into very popular high-level programming language Python 3.

In base the concept of AST (Abstract Syntax Tree) was used and three main components were reflected: Lexer, Parses and Code generator.

Overview of documentation:

- Cosmix
- Code content:
    - Object Generation
    - Lexer
    - Parser
    - Conclusion

Cosmix

Cosmix programming language was invented in 30th of April in 2019 by a young (19-20 years old) and wild (there are less than 40 hours to deadline left) team of developers in Suleiman Demirel University. The main idea of Cosmix language is based on names of datatypes, keywords and variables that are related to the 'Space' and 'Cosmos' theme.

Cosmix Syntax

Datatypes declaration

In datatype declaration in Cosmix languages names of constellations are used

```
virgo - integer
leo - string
libra - Boolean
```

Mathematical operators

In mathematical operators in Cosmix languages are used special words and symbols that are closely related to space theme.

```
§ - range
landedOn - =
travelUntil - <
travelFurtherThan - >
TravelTo - ==
NotTravelTo - !=
```

Keywords

Every programming language has its own keywords to accomplish some tasks. In Cosmix they are also related to out of Earth field. We compare them with Python's keywords

`While` stands for **`For`** loop
`When` stands for **`If`**
`butInCase` stands for **`Else`**
`blackhole` stands for **`False`**
`Space` stands for **`True`**
`Say` stands for **`Print`**

Number Representation

In Cosmix programming language there are special way of number representation

Sun - 0

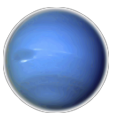
Mercury - 1

Venus - 2

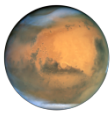
Jupyter - 5

Saturn - 6

Uranus - 7



Earth - 3 Neptune - 8



Mars - 4  Pluto - 9

Complex numbers are written in the form of:

```
0 = Sun
10 = Mercury x Sun
276 = Venus x Uranus x Saturn
```

Code content

Objects generation

All object generation will be done within the objgen.py file and what it simply does is call different objects (classes) which will produce a string of python code and return it.
First of what happens in the objgen.py file is we initialize the ObjectGenerator class with one argument which is source_ast which will be the ordered list of ast dictionaries.

```
class ObjectGenerator():

    def __init__(self, source_ast):
        # This will contain all the AST's in forms of dictionaries to help with
creating AST object
        self.source_ast  = source_ast['main_scope']
        # This will hold the executable string of transplied tachyon code to python
        self.exec_string = ""
```

Next, the `object_definer` method is created which is the main method of the ObjectGenerator class. This method will find all the different ast objects within the ast dictionary and call all the objects and pass in the ast dictionary to get a python string

```
def object_definer(self):

    # Iterate through all ast dictionaries
    for ast in self.source_ast:
        # This will check check if the current AST dict is of which type
        if self.check_ast('VariableDecleration', ast):
            print('var')

        if self.check_ast('ConditionalStatement', ast):
            print('condition')

        if self.check_ast('PrebuiltFunction', ast):
            print('prebuilt')
```

However, within the `object_definer` method there are calls to the `check_ast` which it's role is to check if the current AST's dictionary items being looped through the same key as the `astName` argument passed in

```
def check_ast(self, astName, ast):
    """ Call and Set Exec

    This method will check if the AST dictionary item being looped through has the
    same key name as the `astName` argument
```

```
    args:
        astName (str)  : This will hold the ast name we are matching
        ast     (dict) : The dict which the astName match will be done against
    returns:
        True    (bool) : If the astName matches the one in `ast` arg
        False   (bool) : If the astName doesn't matches the one in `ast` arg
    """
    try:
        if ast[astName]: return True
    except: return False
```

The body generation for a statement is done in the `transpile_body` method which makes use of other methods such as:

- `check_ast`
  - This method will check if the AST dictionary item being looped through has the same key name as the astNameargument to see determine what ast type is being looped through.
- `should_dedent_trailing`
  - This method will check if the ast item being checked is outside a conditional statement.
- `should_increment_nest_count`
  - This method will check if another statement is found and whether or not it should increase nesting count.

The following code snippet is a bit more different as it is generating a for loop which has a statement and body of it's own which both need to be at different indentations therefore it handles the need to decrement when finishing the indentation for a certain statement etc using the methods we spoke about above. Conditional statement, for loops, function declerations all have this same concept applied to them.

```
if self.check_ast('ForLoop', ast):
   if self.should_increment_nest_count(ast, self.ast):
        nesting_count += 1
   loop_obj = LoopObject(ast, nesting_count)
   if nesting_count == 2:
       body_exec_string += "    " + loop_obj.transpile()
    else:
       body_exec_string += ("    " * (nesting_count - 1)) + loop_obj.transpile()
```

Lexer

Data Types

Currently the data types for cosmix are minimal (libra, virgo, leo) and simple so that it is easy to begin with

Using string.split() therefore whenever we come across a DATATYPE word in an item we create a token for it which is as such (for example cases the type is `str`):

```
['DATATYPE', ‘str’]
```

But because of our language has another syntaxes we change it immediately in lever to make work more easier in future

```
DATATYPE = {"libra":"bool", "virgo":"int", "leo":"str"}
```

```
elif word in DATATYPE:
        tokens.append(["DATATYPE", DATATYPE[word]])
```

There are not many keywords (identifiers) for cosmix

```
KEYWORDS = {"when":"if", "while":"for", "butInCase":'else', "space":"True",
"blackhole": "False", "say":"print"}
```

The way we get tokens for identifiers are identical to the way we get the data type tokens file but instead we search through a KEYWORD dictionary instead then create the following token:

```
 ['IDENTIFIER', 'if']
```

Operators

```
["OPERATOR", "+"]
```

The operators cosmix currently support:

- * multiplication
- - subtractions
- / division
- + addition
- % modulus

Code for how we find and generate operator tokens:

```
elif word in "*-/+%":
    tokens.append(["OPERATOR", word])
```

```
We wanted to make our language more closer for humans so we change operator = to
landedOn:
```

```
elif word == "landedOn" :tokens.append(["OPERATOR","="])
```

Comparison

```
 ["COMPARISON_OPERATOR", "!="]
["BINARY_OPERATOR",     "&&"]
```

Comparison Operators

```
elif word == "travelTo" : tokens.append(["COMPARISON_OPERATOR", "=="])

elif word == "notTravelTo": tokens.append(["COMPARISON_OPERATOR", "!="])

elif word == "travelFartherThan" : tokens.append(["COMPARISON_OPERATOR",">"])

elif word =="travelUntil": tokens.append(["COMPARISON_OPERATOR", "<"])
```

Binary Operators

```
elif word == "&&" or word == "||": tokens.append(["BINARY_OPERATOR", word])
```

Integers

As you saw in beginning ,in cosmix we have own counting system.To define which name of planet is number,first of we define is this number is single or it is more complex number by “X” symbol ,to define it we check does it in switcher and does next element is X

```
switcher={"Sun":0,"Mercury":1,"Venus":2,
         "Earth":3,"Mars":4,"Jupiter":5,
         "Saturn":6,"Uranus":7,"Neptune":8,
          "Pluto":9}

 elif word in switcher and source_code[source_index+1] is ‘X':
        source_index+=1
        continue
```

If yes we go to next element ,which will be X and by this we change our number with name of planet to normal symbols

```
elif word=="X":

        first_el=switcher[source_code[source_index-1]]
        next_index=source_code[source_index+1]
        if next_index[len(next_index)-1]=='.':
                second_el=switcher[next_index[0:len(next_index)-1]]
                tokens.append(["INTEGER",str(first_el)+str(second_el)])
                tokens.append(["STATEMENT_END",'.'])
        else:
                second_el=switcher[source_code[source_index+1]]
                tokens.append(["INTEGER",str(first_el)+str(second_el)])
               source_index+=1
elif word in switcher and source_code[source_index+1] is not ’X':
        tokens.append(["INTEGER",switcher[word]])
```

Which look like this

```
["INTEGER", "21"], ["STATEMENT_END", “.”]
```

and for a normal integer is simply: ["INTEGER", “21"]

Score Definer

The scope definers are the opening { and closing } braces which in cosmix are for defining scopes for a function, conditional statement and more. The tokens for scope definers look like this:

```
["SCOPE_DEFINER", "{"]
```

The code for this is super simple just look through a string {} and see if the word being checked is the same as one of the characters in the string.

```
elif word in "{}":  tokens.append(["SCOPE_DEFINER", word])
```

Strings

The following code snippet is a call to the getMatcher method to get the string and return it but what this snippet does extra is check the return to see how to behave.

```
# Identify any strings which are surrounded in  ""
elif ('"') in word:

    matcherReturn = self.getMatcher('"', source_index, source_code)

    if matcherReturn[1] == '': tokens.append(["STRING", matcherReturn[0]]))

    else:


         tokens.append(["STRING", matcherReturn[0] ])
        if ‘.' in matcherReturn[1]: tokens.append(["STATEMENT_END", "."])


        source_index += matcherReturn[2]

     pass
```

End statements

The way we analyze end statements is quite simple and the way we do this is by simply checking every already checked token for a ‘.’ semicolon at the last index of a source code item to see if there was an end statement there. I do this like this:

```
if "." in word[len(word) - 1]:
```

```
                tokens.append(["STATEMENT_END", "."])
```

Parser

Parser class initialiser:

- `source_ast`
  This will initial hold a dictionary with the key of main_scope with the value of an empty array where all the source code AST structures will be appended too. This will be used in order to be interpreted and compiled. The variable will look like this: `{'main_scope': []}`.
- `symbol_tree`
  This will be used in order to store variables with their name and value in the following format ['name', 'value'] so that we can perform semantically analysis and perform checks to see if a variable exists or not.
- `token_stream`

  This will hold the tokens that have just been produced by the lexical analyser which the parser will use to turn into an Abstract Syntax Tree (AST) and Symbol Trees so that syntactic and semantical analysis can be performed.

- `token_index`

  This holds the index of the tokens which we have checked globally so that we can keep track of the tokens we have checked.

`parse()`

This will parse the tokens given as argument and turn the sequence of tokens into abstract syntax trees.

Arguments

- `token_stream (list)`
  - The tokens produced by lexer

Returns

- `source_ast (dict)`
  - This will return the full source code ast

This method tries to identifiy a pattern of tokens that make up a parse tree for example a variable would be recognised if the parse method stumbled across a datatype token

`(['DATATYPE', 'str'])`

it would know it is a variable decleration and call the `variable_decleration_parsing()` passing in the token stream from where the data type was found with the rest of the tokens and will also pass in false because this parsing isn't called from a body statement parser. This is how it looks in code:

```
if token_type == "DATATYPE":
```

```
  self.variable_decleration_parsing(token_stream[self.token_index:len(token_stream)],
False)
```

This is then repeated for for all the tokens in the source code to find patterns and create AST's from them. Once all this is done the method checked for an error messages like such:

```
if self.error_messages != []:
    self.send_error_message(self.error_messages)
```

This checks if the error message array is empty and if so no errors occurred during parsing but if there is then error messages will all be displayed in hierarchy order from first to last. Finally, the method then returns the `source_ast` so that it can then be used by the ObjectGenerator (`objgen.py`) to make the Objects for the different AST's and transpired them into python.

---

variable_decleration_parsing()

This method will parse variable declerations and add them to the source AST or return them if variable decleration is being parsed for body of a statement.

Arguments

- `token_stream (list)`
  - The token stream starting from where the variable decleration was found
- `isInBody (bool)`
  - This will declare whether the var decleration is being parsed within a statement body.

Returns

- `ast (dict)`
  - The condtion ast without the body
- `tokens_checked (int)`
  - The count of tokens checked that made up the condition statement

---

conditional_statement_parser()

This will parse conditional statements like 'if else' and create an abstract sytax tree for it.
Arguments

- token_stream (list)
  - Tokens which make up the conditional statement
- isNested (bool)
  - **True** the conditional statement is being parsed within another conditional statement **False** if not.

What this method does is handle the parsing of conditional statements by tying in all the methods needed for thise which are `parse_body` and `get_statement_body`. The way conditions and the body of that conditional statement is parsed is separate .

The way the condition is parsed is simply by looping through the conditional statement getting the `first_value` followed by the `comparison_type` and finally the `second_value`. Once we got that the for loop should get a opening scope definer ({) which will then break and handle the parsing of the body seperately which is explained in more detail by the `parse_body` and `get_statement_body` documentation.

Conclusion

Thank you for your attention and all knowledge that we gained in this course. We believe that you can't be a real programmer if you've never written your own compiler ☺️