Domain Specific Language: FinancePY for manage basic group accounts.

Jean Carlo Camacho Espín
School of Mathematics and Computational Science
Yachay Tech University
Urcuquí, Ecuador
jean.camacho@yachaytech.edu.ec

***Abstract*—The design of programming languages to solve certain problems is the core of the development behind them. And the domain-specific languages (DSL) are generally optimized to solve particular problems in easier or optimized ways. This work dives into the problem of group finances where people can handle this with the generation of documents that contain the data of the accounts. Also, this works carry concepts behind the creation of programming language and proposes a DSL (FinancePY), and acknowledges the process of its creation.**

***Index Terms*—DSL, finances, programming languages, Python, economy**

I. Introduction

It is possible to use a Python script to manage a .csv or .txt file. But there are specific languages optimized for managing databases, such as MySQL or MongoDB that can perform the same task more efficiently, with better features to modify the databases, etc. That is the reason a domain-specific language (DSL) is useful when performing a specific task.

Some well-know DSL languages are HTML (web pages), CSS (styling web pages), MySQL (relational databases), Latex (create pdf documents), etc. However, there is not any DSL specialized in financial works. The objective of this work is to propose a DSL model based on Python that can generate and manage .csv files that will carry the information of the finance of a group and will allow doing some operations to visualize the document information and will allow to modify it as well.

The importance of a DSL focused on economy is that, the script actually could act as a second register of the different movements, the transactions, the deposits, the account settle, the users and the amount of money and items that are added, all this information will act as a register, and at the same time this DSL will give some functions for a better visualization of certain characteristics, such as the visualization of the data of a specific user, or the information of those who have money, those who are in zero or in negative values, etc.

II. Related works

There are not recent works published related to the creation of programming languages or DSL. However, works about a programming language that has more than ten years of age were with important information and concepts of how to structure and create a programming language, such as the work of Martín [2].

Also in this works the authors show useful tools with which they created their programming languages as in the work of Herbert *et. al.*. [4]. Also works where the authors purpose solutions to very specific problems with a DSL as in the work of Marcin and Kazimierz [3], Junwei, Zhongxiang and Yujun [6] and Enes *et. al.*. [1]

III. Methodology

Programming languages have a similar pipeline of construction [5]. It is necessary to think in the first place about the domain of the language. In the case of ”FinancePY”, the language's main idea is to manage group accounts. A characteristic it must have is to store information and be able to add/change that information based on the transactions and different operations.

This file must have a standardized methodology of creation, such that the language will not have any problems while reading and writing over it. A possible structure generation could be a file separated by sections, like the header that can carry the filename, the users, the money each user has, and a Boolean field saying if the account is settled between the current users and until the last transaction.

On the other hand, the language shall have operations to interact with the stored data, as it could be the option to add more money to a user. Allow to add more items to the inventory, and discount the money from the user account that is adding that item to the list. Search if a user is part of a specific group and every item that the user added to the account with the respective description and prices. Also a function to settle the account, in such a way that everyone gets a balanced account. Some users will get negative values; others will get an increment on their balance, depending on how much they spent on the items they bought.

This kind of functions modify directly the account file, however other functions will only read the data and interpretate it to generate an output. It could be possible to ask for the users with a negative or positive balance on their accounts, and also who has zero money. Search for the data of only a specific user, and print the transactions than that user preformed.

Those are only general ideas of the possible functions the DSL could have implemented. Considering this, it is possible to get familiarized with the creation process of a programming language. As we can see on the Fig. 1. There are three main

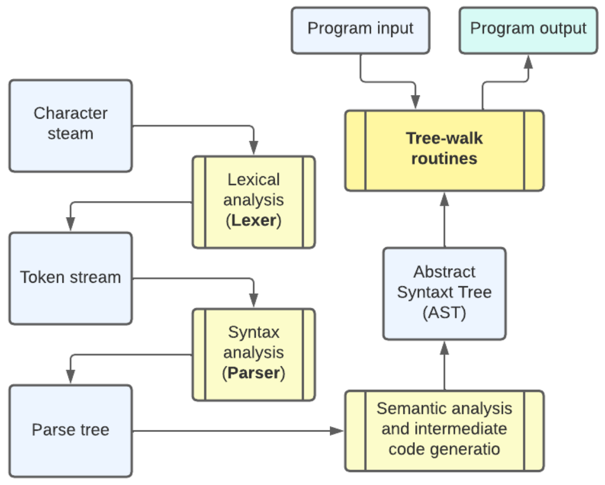


Fig. 1. Diagram that shows the phases a interpreted language passes through to get an output

```
tokens.ignore(r'[ \t\r\f\v]+')
tokens.ignore(r'#[^\n]*\n')
tokens.ignore(r'\n')

tokens.add('SETTLE_ACC', r'SETTLE_ACC(?!\w)')
tokens.add('BALANCE', r'BALANCE(?!\w)')
tokens.add('ITEM', r'ITEM(?!\w)'
tokens.add('NEW_GROUP', r'NEW_GROUP(?!\w)')
tokens.add('READ_GROUP', r'READ_GROUP(?!\w)')
tokens.add('ADD_USER', r'ADD_USER(?!\w)')
tokens.add('INVENTORY', r'INVENTORY(?!\w)')
tokens.add('DEPOSIT', r'DEPOSIT(?!\w)')

tokens.add('STATE_TOKEN', r'(zero(?!\w))
tokens.add('STATE_TOKEN',r'(positive(?!\w))
tokens.add('STATE_TOKEN', r'(negative(?!\w))')
tokens.add('ID_TOKEN', r'("[a-zA-Z_\s]+[a-zA-Z](?!\w)")
tokens.add('ID_TOKEN', r'(\'[a-zA-Z_]+[a-zA-Z](?!\w)\')')
tokens.add('STRING', r'(".?")||(n'.?n')')
tokens.add('MONEY', r'(\d+.\d\d)')
tokens.add('MONEY', r'(\ d+.\ d)')
tokens.add('MONEY', r'(\d+)')
tokens.add('BOOLEAN', r'(True)||(False)||(true)||(false)||(T)||(F)||(t)||(f)')

tokens.add('SEMICOLON', r';')
tokens.add('COMMA', r',')
tokens.add('(', r'\(')
tokens.add(')', r'\)')
```

TABLE I
LIST OF THE RECOGNIZED AND IGNORED TOKENS

steps, the lexical analysis, the syntax analysis and the semantic analysis and each process needs something from the previous step, the input to the flowchart is the character steam, that is basically the code with the instructions.

A. *Lexical analysis (Lexer)*

On the lexical analysis, we must provide a manner for the computer to interpret the words that constitute the code. To create the lexer in this work it will use the RPLY library from Python, which works as an extension of the PLY library, which handles the lexer and the parser for Python.

In the lexer, it is important to tell the program which tokens to recognize, and which tokens to ignore. Once is defined a variable as the lexer generator, we add to it the tokens and how they will be recognized.

In Fig. I. we can see that the lexer will ignore the blank spaces, and the parser will generate less complex syntax trees, in consequence the process will be faster, the lexer will ignore any string that starts with a "#" until the end of the line, those are the comments. On the other hand, this language does not handle cycles or functions that need any indentation type, so the blank spaces do not get any functionality.

In the table, we can see the definition of the "ID_TOKEN" token, which can be any string with letters from "A" to "Z", mixing capital letters, lower-cases, and underlines to generate the IDs. This token will represent the user ID and the account ID, depending on the command, during the evaluation the decision to process it as a user or account ID, will be based on the command token. Also, there is a "STRING" token, that accepts any string inside a pair of quotation marks, this will be used to tag the items passed to the inventory. The "MONEY" token will accept any positive integer or floating number that has as many as two decimals, to represent integers and cents on the money. Also, the language accepts a "BOOLEAN" token that will indicate if the bought item has taxes included on the price, this token is optional and if it is not called, the value is false by default.

Finally, the "STATE_TOKEN" token, is represented by 3 words; "positive", "negative", and "zero". Also, some structure punctuation symbols are used, after calling a function token, it is expected to evaluate the parameters inside parenthesis "(" and ")" and at the end is expected a "SEMICOLON" token to determine the end of the command and line, also the parameters inside are separated by "COMMA" tokens.

On the other hand, there are a few function tokens that will execute different functions; The "SETTLE_ACC" will call an operation to balance the internal account money, dividing fairy the spent bill between all the members in the account at that moment. The "BALANCE" function will receive as input a "STATE_TOKEN" token and depending on which token it receives will display the user's information of those whose money balance is "positive", "negative" or have no money ("zero"). The "ITEM" function will let introduce a new item, with the data of the user that bought it, the item name, the value, and optionally if it carries taxes. The "NEW_GROUP" will generate an empty new group file. The "READ_GROUP" will charge on memory the name of a group account, so the other functions will know to which account executes their functions. The "ADD_USER" will add a user with some money to an account. The "INVENTORY" function will show the items bought by the user ID passed as a parameter. Finally, the "DEPOSIT" function will add money to a specific user.

B. *Syntax analysis (Parser)*

Once the lexer separates the string into tokens, it is possible to create a parser that produces the parse tree. The parser gives sense to the generated tokens that into the syntax tree get different meanings for the execution based on the context of the whole syntax tree.

But to build the parser is mandatory to generate functions that represent a general view until it reaches every process that is represented by the tokens. And the boxes will have a context-free grammar that represent how each expression will behave on a brief way. The boxes code in Python with the RPLY library and the BaseBox module, it is proposed the next distribution of boxes for the parser.

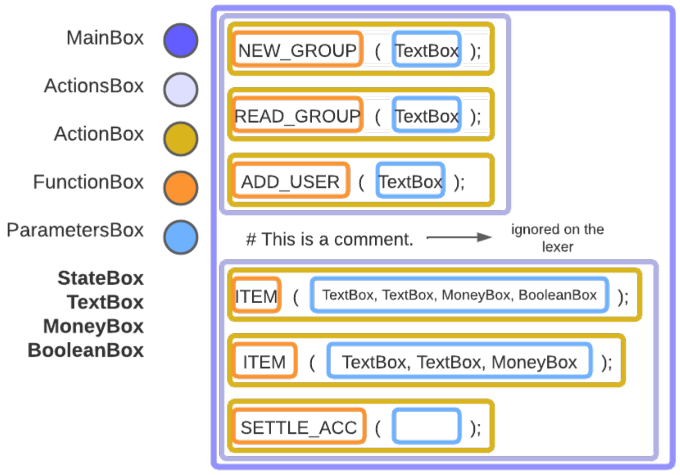


Fig. 2. Proposed distribution of the boxes the language will recognize

In Fig. 2. we can appreciate the proposed distribution, where the ”MainBox” is the box that calls the ”ActionsBox” once, then the ”ActionsBox” can get more ”ActionsBox” or ”ActionBox” or nothing. The ”ActionBox” contains the ”FunctionBox”, and ”ParametersBox”, it also includes the punctuation tokens as ”(”, ”)” and ”SEMICOLON”, the punctuation tokens are not evaluated. However, are needed to follow the command structure. The ”FunctionBox” will receive any of the functions tokens, and the ”ParametersBox” will get different combinations of variables, that will be selected depending on the ”FunctionBox”, but this will always return all the possible variables, even if it is not used it will return a ”None” value. On the other hand, the context-free language that gives sense to the BaseBox content gets defined in Table II. It represents Fig 2. language. Nevertheless, the functionality of each function is not defined in this context-free language however, will be implemented in Python, and the process explanation will be with the implemented code.

C. Semantic analysis and code generation (Abstract Syntaxt Tree)

At this stage, we already have a grammar, a lexer, and a parser based on the BaseBox library from RPLY, It was described in the previous section with low detail, how the final structure will be given, however on this section, the description of the abstract syntax tree (AST) will be represented more clearly.

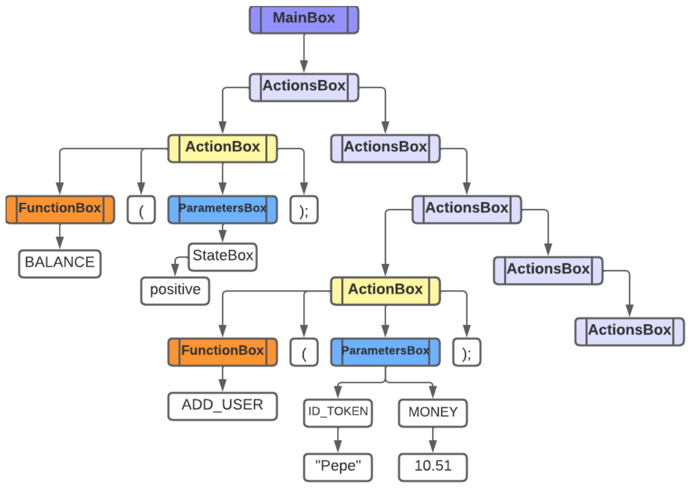


Fig. 3. An example of an abstract syntax tree of a two rows script

In Fig. 3 is a representation of how the abstract syntax tree develop itself, and how the ”ActionsBox” calls itself to keep adding more rows as necessary. On the graph is shown the evaluation the program takes and what symbols are ignored on the Tree-walk routines.

With the functions built on the BaseBox.py file, the production rules will be set. As previously said, there will be a ”mainBox” rule that will receive as input an ”actionBox”, and this will call the ”MainBox” function, passing the ”actionsBox” as a parameter. The ”actionsBox” will have two structures, in the first, it can be passed another ”actionsBox” with an ”actionBox” and this will call the ”ActionBox” function and will receive both as parameters; In the second it can receive only an ”actionBox”, that will call the same function but in the space where an ”actionsBox” is sent, it will be sent ”None”.

There are productions rules that will call the evaluation of the parameters, those are the ”stateBox” that receives a ”STATE_TOKEN” and will call the evaluation on the ”StateBox”, a similar process happens with the ”textBox” that can receive a ”STRING” or an ”ID_TOKEN” that will be evaluated with the ”TextBox” function. The ”moneyBox” receives the ”MONEY” token and returns its evaluation on the ”MoneyBox” and the same is with the ”booleanBox” that receives the ”BOOLEAN” token and returns the evaluation in the ”BooleanBox” function.

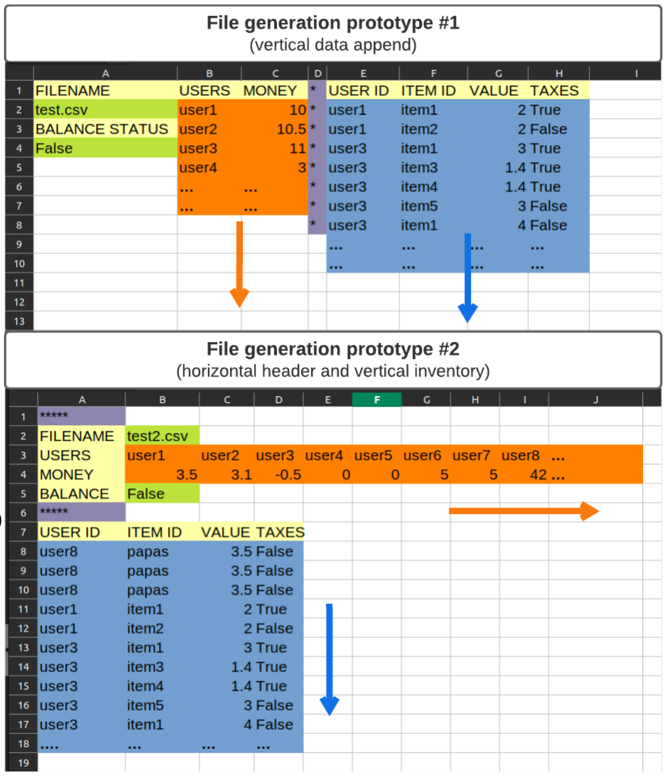
The ”functionBox” will have a list of all the possible function tokens, it will evaluate one at a time and will return its evaluation on the ”FunctionBox” function. The ”actionBox” will receive this next structure: ”functionBox” (”parametersBox”) ”SEMICOLON” from where the ”actionBox” returns a single value and the ”parametersBox” return an array that contains the values of the parameters, if from a specific function there is no parameter, it will be filled with a ”None” and this will call the ”ActionBox” function that receives 6 parameters, the first is the ”actionBox” evaluation, and the other 5 are each of the parameters as it matches.

The ”parametersBox” can have multiple forms, It can receive a single ”stateBox”; a single ”textBox”; an chain of ”textBox”, ”COMMA”, ”textBox”, ”COMMA”, ”moneyBox”, ”COMMA”, ”booleanBox”; a similar chain as the previous, but without the last ”COMMA”, and ”booleanBox”; a chain with a ”textBox”, ”COMMA”, ”moneyBox”; or even an empty chain.

CONTEXT FREE GRAMMAR

string → (”[a-zA-Z] $*$ $[a - zA - Z] *$ ”)
numbers → (0 $\|1\|2\|3\|4\|5\|6\|7\|8\|9) *$
acc_id → (string)
usr_id → (string)
item_id → (string)
MoneyBox → (numbers)*.(numbers)*
BooleanBox → ((True, T, t) $\|(False, F, f))$
MainBox → (ActionsBox)
ActionsBox → (ActionsBox) (ActionBox)*
ActionBox → (FunctionBox) ”(” (ParametersBox) ”)” ”;”
ParametersBox → (stateBox) $\|(textBox)\|(textbox",\text{"}moneyBox)$
ParametersBox → (textBox ” , ” textBox ” , ” moneyBox ” , ” booleanBox^{+})
textBox → ((ID_TOKEN) $\|(STRING))$
FunctionBox → (SETTLE_ACC, ADD_USER, ITEM)
FunctionBox → (SEARCH_U, DEPOSIT, READ_GROUP)
FunctionBox → (NEW_GROUP, BALANCE, INVENTORY)

TABLE II
A FORMAL DEFINITION OF THE CONTEX-FREE GRAMMAR, EXPRESSIONS DEFINITIONS WILL BE DETAILED FURTHER AHEAD.



File generation prototype #1
(vertical data append)

	A	B	C	D	E	F	G	H	I
1	FILENAME	USERS	MONEY	*	USER ID	ITEM ID	VALUE	TAXES	
2	test.csv	user1	10	*	user1	item1	2	True	
3	BALANCE STATUS	user2	10.5	*	user1	item2	2	False	
4	False	user3	11	*	user3	item1	3	True	
5		user4	3	*	user3	item3	1.4	True	
6		...	...	*	user3	item4	1.4	True	
7		...	...	*	user3	item5	3	False	
8				*	user3	item1	4	False	
9					...	...	...	...	
10					...	...	...	...	
11									
12									
13									

File generation prototype #2
(horizontal header and vertical inventory)

	A	B	C	D	E	F	G	H	I	J
1	*****									
2	FILENAME	test2.csv								
3	USERS	user1	user2	user3	user4	user5	user6	user7	user8	...
4	MONEY	3.5	3.1	-0.5	0	0	5	5	42	...
5	BALANCE	False								
6	*****									
7	USER ID	ITEM ID	VALUE	TAXES						
8	user8	papas	3.5	False						
9	user8	papas	3.5	False						
10	user8	papas	3.5	False						
11	user1	item1	2	True						
12	user1	item2	2	False						
13	user3	item1	3	True						
14	user3	item3	1.4	True						
15	user3	item4	1.4	True						
16	user3	item5	3	False						
17	user3	item1	4	False						
18	....	...	...	...						
19										

Fig. 4. Comparison between the two prototypes of .csv file generation

IV. FINAL IMPLEMENTATION

1) File format: The actual prototype is just Python code, where its implementation was with Python functions only. The first thing I defined before writing the code was the file type where the accounts data will be saved. Python can manage .txt files, but the CSV library gives more facilities to separate the data due to the separation with commas. Reading and writing a .csv file requires less code, and the .csv format could be handy in another application.

For the selected data structure, there was two ways to generate the files and different methods for searching the data and write.

2) CSV file structure: As shown in Fig. 4. the first prototype generation had the header on the first line, except the balance status that was in the first column, below the file name, then the users and their money data can be appended at the bottom of the list, and the same with the inventory. Nevertheless, the reading and writing of the file are made by rows, and the growth of users and items was not equal, so appending the data or changing the values needed more complex methods to modify the file.

On the other hand, the second prototype generation generates the header horizontally, so the first row of the header is only the filename. The users and their money have a row that is easy to read because a single row contains a single data type (only the users, only the money, only the balance status). The users and money rows will grow to the right. For the inventory structure, the data grows down, but each row will contain only one transaction information, separating any header or user data from the transactions is not needed.

3) Functions: The functions prioritized for the implementation were; the file generation function (NEW_GROUP), the file reading function (READ_GROUP), the add user function (ADD_USER), and the add item function (ITEM). This set of functions makes it possible to create a new account, add items and users and read files. This is the base of the programming language because it does not makes sense to build operation functions if there is no way to read/store the data.

The function ”NEW_GROUP” receives an ”ID_TOKEN” as input and checks if there is not an existing file with that ID; if not, it generates a formatted file without users, money, and items, and the account balance on ”true”. The function returns the used ID, making it possible for other functions to access the last file used.

The ”READ_GROUP” function receives an ”ID_TOKEN” as input, it checks in the current directory for any CSV file with that ID. If it exists, it stores that name on a global variable from where other functions will get the ID. Else it returns an error message.

The ”ADD_USER” function receives the ”ID_TOKEN” and ”MONEY” as input. The function will check for an account ID on memory and that the user ID is not stored on the file already. If there is no problem, the function will append the user and the money at the end (to the right) of the user’s list and sets the account balance to ”false”.

The ”ITEM” function receives two ”ID_TOKEN”, where the first is a user ID and the second is a string that represents an item ID, the money token, and the Boolean token. This token by default is ”false” if it is not specified. The function will check the account ID on memory and that the user ID exists on that account. If there is no problem, the function appends at the end of the file the new transaction, which will be the next space of the transactions list, and sets the account balance on ”false”. Also if the taxes Boolean is true, the value is registered, adding the taxes value (12%), if not it registers the value as it is in the input.

There are also other functions that are less essential but were also implemented on the DSL FinancePY. There are the ”DEPOSIT”, ”INVENTORY”, ”BALANCE”, ”SETTLE_ACC” functions.

The ”DEPOSIT” function receives a user ID and a money amount, the function checks if there is an account ID on memory and searches for the user that matches with that ID and adds the money.

The ”INVENTORY” function receives the user ID and returns all the items that were bought by the same user. The function checks if there is an account ID on memory and prints the required data.

The ”BALANCE” function receives a ”STATE_TOKEN” that could be ”zero”, ”positive” or ”negative” and depending on the token it returns all the users in the account that have zero money, a positive balance, or a negative balance as it corresponds. Also, this function checks if there is an existing account ID charged on memory.

Finally, the ”SETTLE_ACC” balance the money each user has in such a way that all the users pay equally the amount of money that was spent in the inventory, from all the items where the paid flag is false, the money is added to calculate the total spent money, which is divided between the number of users, and finally, this value is subtracted from the actual amount of money each user has and is added the amount of money they spent individually. This balance of the account and the paid flag change to true.

V. Conclusions and future work

It was deployed a DSL language created with Python and its library RPLY that handles the management of basic group economy. However there are a lot of possibilities in the language, a lot more operations can be implemented, and even the interaction between different account files could be an option. However, for the scope of this work, the main goal was reached, and is highly recommended to the reader to test and improve this work, by accessing the main source code on GitHub.

References

[1] Enes Altuncu et al. “Blocks and text integration in a language-based editor for a domain-specific language”. In: *2017 International Conference on Computer Science and Engineering (UBMK)*. 2017, pp. 1084–1089. doi: 10.1109/UBMK.2017.8093484.

[2] Martin Fowler. “A Pedagogical Framework for Domain-Specific Languages”. In: *IEEE Software* 26.4 (2009), pp. 13–14. doi: 10.1109/MS.2009.85.

[3] Marcin Kowalski and Kazimierz Wilkosz. “A Domain Specific Language in Dependability Analysis”. In: *2009 Fourth International Conference on Dependability of Computer Systems*. 2009, pp. 324–331. doi: 10.1109/DepCoS-RELCOMEX.2009.14.

[4] Herbert Prahofer et al. “The Domain-Specific Language Monaco and its Visual Interactive Programming Environment”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. 2007, pp. 104–110. doi: 10.1109/VLHCC.2007.14.

[5] Michael L. Scott. “Chapter 2: Programming Language Syntax”. In: *Programming language pragmatics*. 4th. ELSEVIER, 2016, pp. 43–102.

[6] Junwei Yang, Zhongxiang Hu, and Yujun Zheng. “Macml: A Domain-Specific Language for Machinery Service Management”. In: *2010 International Conference on Service Sciences*. 2010, pp. 293–297. doi: 10.1109/ICSS.2010.12.