Group 13
CS3020
7 October 2015

lispy2.py

A scheme interpreter in python

(An ((Even Better) Lisp) Interpreter (in Python))

CONTENTS OF REPORT:

INTRODUCTION:

In the previous semester we looked at a version of lispy, namely lisp.py. The main objective was to get an overview of the general procedure on how the interpretation process takes place. In this assignment we take a look at a more intricate interpreter; accommodating a wider range of cases in scheme. lispy2 is an augmentation of lispy, which is almost three times as complex. In the following report we'll be looking at this new code with much more depth.

INTERESTING ASPECTS OF THE LANGUAGE:

Scheme being a functional programming language makes it unique in several ways. While traditional object-oriented and scripting languages are rigid in the sense of data types and structure, Scheme holds its power within lists. Scheme is a dialect of LISP; each program constitutes of a series of lists, nested or linked to one another dynamically. The speciality of these lists are that they can contain data from a range of different data types, may that be keywords, strings, bools, or even complex

numbers. This makes the parsing and semantic phase intensely complex; giving scheme a unique interpreting methodology.

Another interesting thing about Scheme is that the AST generated from the parsing phase, is in correlation with the actual structure of the program. This makes sense, as compilers for various languages do create their AST's in the form of nested lists.

Lambda calculus is essential to Scheme, providing the programmer with immense support, especially in a language where implementations are mostly recursive. Care has to be taken in the parsing phase of this language as many function calls and lambda definitions have different number of arguments, which is again essentially an embedded list.

THE LISPY2.PY:

LISPY2.PY was chosen due to its well defined divisions between different phases of the interpreter. Each segment like a normal compiler has 5 broad phases. The phases here included in this interpreter were lexical analyser, syntactic analyser, and semantic analyser. While each section was interrelated and present within the same code, each phase was implemented uniquely. Unlike lisp, where the lexical analyser used to generate tokens by splitting up the code based on spaces, the presence of string definitions force the LA to apply a more traditional approach with the use of regular expression.

Another interesting aspect of this interpreter is that unlike the conventional interpreters the implementation of the syntactical analyser is more complex than that of the semantic analyser. This is because the parser has a harder job; it has to not only make sure that legal code has all the right pieces, but deal with illegal code, producing a sensible error message, and extended code, converting it into the right basic form.

Explanation for both the statements above will be given in the next section.

THE LEXICAL AND PARSING PHASE:

THE LEXICAL PHASE:

Unlike the previous version where the tokens were generated from a string, this interpreter accommodates command based input, spanning over multiple lines, with proper error handling.

The token generator is based on a regular expression, enforced by the python regular expressions. Each input is taken from the input stream and separated into tokens using the next_token class method (in the class InPort). Each token is first checked for EOF. If the test passes (i.e. There is no EOF), the input stream is tokenised and returned (except the empty string). All comments (tokens beginning with ";") are discarded in this phase.

The reason for such an implementation (apart from the obvious error handling) is that the language construct includes string as a data type (as opposed to the previous implementation with just symbol, int, and float). Therefore the naive method of tokenising on the bases of the space delimiter can no longer be used.

THE PARSING PHASE:

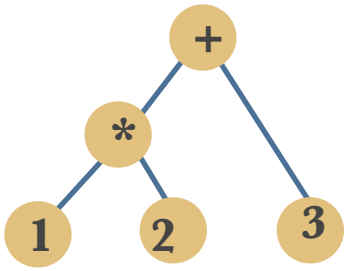
As mentioned earlier, the parse tree that is generated from the parser, is exceptionally close to the actual structure of the program itself. Therefore, lispt2.py uses python lists to map the corresponding Scheme lists. Each python list is created when the parser encounters the "(" token and terminated with ")". The tokens within these parentheses are added onto the list in a recursive fashion. Error handling such as EOF and unmatched parentheses is taken care of in this phase.

Since Scheme consists of only lists, care must be taken whether the given list is to be evaluated (in turn if it to be parsed into a tree as separate), or is the list itself an object that is to be operated on and treated as such. Such lists are prepended by a "quote", and are returned as a list of the token "quote" and the corresponding list.

The data types of Scheme are trivial, and a regular condition construct (atom, and to_string methods) are used to convert the scheme object into its corresponding data type and vice-versa.

An interesting feature of Scheme is that is has a prefix notation; removing the burden of taking into consideration precedence. The tree constructed is based purely on the structure of the code itself, with the parent at each step being the first object defined. Example:

(+ (* 1 2) 3) will be constructed as:



One of the main conflicts that the parser has is to maintain the same semantic for various constructs of the syntaxes. Let us take the if condition as our example. For many standardised languages, the rule that is often chosen for if constructs is that the if maps onto the first else. Scheme does not follow that rule, therefore the interpreter must normalise both type of if conditions (if and if then else) in order to validate the program. It does so by the means of expansion. It places an additional list representing "none", in order to complete the if then else construct. The following is the example taken from the documentation:

Extended Expression	**Expansion**
`(begin)`	`None`
`(if` *`test conseq`*`)`	`(if` *`test conseq`* `None)`
`(define (`*`f arg...`*`) `*`body...`*`)`	`(define` *`f`* `(lambda (`*`arg...`*`)` *`body...`*`)`
`(lambda (`*`arg...`*`) `*`e1 e2...`*`)`	`(lambda (`*`arg...`*`) (begin `*`e1 e2...`*`))`
`` ``*`exp`* `(quasiquote `*`exp`*`)`	expand `,` and `,@` within *exp*
`(`*`macro-name arg...`*`)`	expansion of (*macro-name arg*...)

The parsing phase takes care of this expansion in the function expand. Here it checks the first element of the list and matches it with the

corresponding if condition. After which it validates the parsing by checking that the list has the correct number of arguments, and that certain data elements are of the correct data type. It is here that the expansion takes place. It treats the extended expression as its corresponding expansion with the help of indices.

INTERESTING ASPECTS OF THE INTERPRETER:

The semantic analysis was not highlighted to a great extent in this report, however one of the most intriguing aspects of this implementation that needs to be highlighted is the tail recursion optimisation. Since iteration is not an option in Scheme, even the most basic loops have to be implemented recursively. Hence a huge burden is placed upon the stack. In situations where more than 100 frames need to be used to store information on the stack, it is highly likely for the stack to run out of memory.

The work around for this in the lispy2.py is to make modifications to the variables and environments, as long as the end case is not called explicitly. This would avoid having to create multiple frames and created separate instances of the same variable.

REFERENCES:

http://ds26gte.github.io/tyscheme/

http://norvig.com/lispy2.html