Satya Vasanth Reddy - CS13B1033
Arjun V Anand - CS13B1041
7 October 2015

ANALYSIS OF LEXICAL AND PARSING IMPLEMENTATIONS

HTML & LISP

HTMLParser

Source: https://hg.python.org/cpython/file/2.7/Lib/HTMLParser.py

Why HTML ?

HTML is one of the most user friendly and error tolerant language.It is more expressive than another language.There aren't many reserved keywords.The Grammar rules are simple too.

The HTMLParser

The report is about the usage of a python tool for html parser . This interpreter has lexical and the parsing components combined.Most of the handle decisions are overridable which gives more flexibility to the further interpreting stages.

The advantage in HTML parsing strategy is, that even though there are some stray sequence of characters , it wont effect the total parsing . This property is mainly because of the too few grammar rules in HTML .

Each and every method has a check provided to ensure if it is correctly called.

e.g.:assert rawdata[i:i+2] == "</" in parse_endtag..

Handling Exceptions:

The HTMLParser has a class HTMLParseError for handling the errors.It uses the lineno and offset attributes of the HTMLParser class and displays appropriate location of errors.

Suggestions to the implementation:

Since most of the handling cases are to be overridden, the parsing of start and end tags can be done using a stack where we push a start tag into the stack and when we come across an end tag of the same type of tag on the top of the stack we pop it. If it's not same , error has to be reported.

Topics covered in report:

The report explains in detail the implementation details and the function of the parser and the reason behind the implementations.

Tokens

NAME	REGEX	PURPOSE
interesting_normal	`[&<]`	Default search to start with
incomplete	`&[a-zA-Z#]`	Incomplete tags
entity ref	`&([a-zA-Z][-.a-zA-Z0-9]*)[^a-zA-Z0-9]`	Reference to entities
charref	`&#(?:[0-9]+|[xX][0-9a-fA-F]+)[^0-9a-fA-F]`	Reference to characters
opentag	`<[a-zA-Z]`	Tag Openings
closetag	`>`	Tag Closings
comment close	`'--\s*>'`	Comment closing
tagfind	`([a-zA-Z][^\t\n\r\f />\x00]*)(?:\s|/(?!>))*`	Finding a tag name
attrfind	`r'((?<=[\'"\s/])[^\s/>][^\s/=>]*)(\s*=+\s*'r'(\'[^\']*\'|"[^"]*"|(?![\'"])[^>\s]*))?(?:\s|/(?!>))*')`	Finding the attribute part of a tag

locatestarttagend	`<[a-zA-Z][^\t\n\r\f />\x00]*`
	`(?:[\s/]*`
	`(?:(?<=['"\s/])[^\s/>][^\s/=>]*`
	`(?:\s*=+\s*`
	`(?:'[^']*'`
	`|"[^"]*"`
	`|(?!['"])[^>\s]*`
	`)`
	`)?(?:\s|/(?!>))*`
	`)*`
	`)?`
	`\s*`
		Locates the end of a start tag .Includes both simple tags and tags with attributes

endendtag	`>`	Ending of an end tag
endtagfind	`</\s*([a-zA-Z][-.a-zA-Z0-9:_]*)\s*>`	Finding the closing tag

?:Match expression but do not capture it ?! Match / if not followed by > (?<=abc)def will find a match in abcdef

The Python Parser uses 2 modules `markerbase` and `re`
The following are the methods used from `markerbase.py` in the program

`reset(self)`	The self variables ,line number is set to 1 and the offset is 0
`updatepos(self,i, j):`	Updates the offset and returns the last found index of newline

Usage :

```
p=HTMLParser()
p.feed(htmlfile)
```

• As soon as an object of HTMLparse is created the `__init__` calls the reset method which initialises the attributes

```
self.rawdata = ''
self.lasttag = '???'
self.interesting = interesting_normal
self.cdata_elem = None
markupbase.ParserBase.reset(self)
```

• Parsing a HTML file is done by using the feed function

• The feed method adds the data to the `rawdata` attribute of the parser object and calls the `goahead` method

```
def feed(self, data):
self.rawdata = self.rawdata + data
self.goahead(0)
```

• In `goahead,` the loop runs over the `rawdata` .It first searches for an '&' or '<'.

• If there is a match found, the start index of the match is stored in j else j becomes n .So the remaining unparsed data is passed to handle_data. and the offset and linen are updated and we break from the loop.Now end is 0 initially so it ends the parsing by changing the rawdata attribute of the self object.

•

• But if we find a match of < or & then,we calculate the first occurrence and call the handle_data over the portion between current offset to the first occurrence and the offset is updated.

•

• If it starts with '<',it can be a `starttag / endtag / comment /pi /declaration` if it matches nothing, we simply skip "<".

• If there is a match with `starttagopen`, the method `parse_starttag(i)` is called :

• `parse_starttag(int i)`

• It calls the method `check_for_whole_strt_tag().`

• This function locates the match from beginning to end of a starttag starting from a given index i. Depend on the following strings > or '/', we update and return the appropriate value such that the `parsestarttag` starts parsing from the next char.

• -1 is returned for inappropriate `starttags` from both the functions.

• Since we know the position where the start tag ends, the start tag is stored in `self.__starttag_text`

• Now, we initialise the empty array of attributes.

• There can be variable number of attributes in the start tag. We start with i+1 index and find the name of the tag. The end of tag is already stored in `endpos`. So from the end of tagname to the end of starttag, we iterate through and find the attributes

• For each attribute match,we get 3 group items name rest and attribute. If the rest is null, we assign the value of attribute to be None and also trim off the " and ' in the attribute values.

• A method unescape is used to remove the special character quoting and update the attribute values.

• The K value is then set to the end of the current match.

• The remaining portion of start tag without whitespaces either ends with /> or >. So the methods `handle_startendtag(),handle_starttag()` parses them respectively.

• If the tag is either a script or a style, the `cdata_mode` is converted to script or style mode respectively .

• The cdata_mode is used in further parsing .

• **parse_html_declaration**()

• It checks if the prefix is '<!' and proceeds to find if the rawdata from i is a comment or a bogus comment or an actual html declaration.In case of `'<!doctype'` which is case insensitive, the encloseing '>' is found and the substring is passed to handl_decl.

• In case of bogus comments,the `parse_bogus_comments()` checks if the rawdata from index i starts with </ or <! and searches for the immediate next >.

• If the closing > is found it is passed to handle_comment else -1 is returned.

• `parse_endtag` first looks ahead if the following string is "/>" .In this case it returns i+3 else it looks for the '>' occurrence and for the successful match, the the tag name is passed to `handle_endtag()` and the `match.end()` is returned.Unless and until the `cdata_element` is not null the tag data in the end tag is passed to handle data. Finally, since we are out of that tag we clear the `cdata_mode` .

• Similarly, the **parse_pi** also searches for the closing > and returns the closing index which is stored in k.

• Now the `offset i` is updated to k using the `updatepos()`

• .Suppose, if the rawdata from i doesnot start with '<' .But instead starts with '&#',we know that its a charecter reference.The corresponding match is sent to handle_charref

This function handles the case if ; is missed.

```
if not startswith(';', k-1):
k = k - 1
i = self.updatepos(i, k)
```

• But if we find no match for `charref` and find a semicolon in the data, &# is consumed without doing anything and break from the loop.

• If it startswith '&' and we find a match for entity reference, we update the position depending on whether we see a semicolon or not.

• If there is no match for entity reference, we check if it matches with the incomplete regex .If it matches, an error is raised

```
self.error("EOF in middle of entity or char ref")
```

Or if & is the last charecter of the file, & is consumed and the position / offset is updated.

- As a whole start tags are handled by calling `self.handle_starttag()`or `self.handle_startendtag()` and end tags by `self.handle_endtag()`. The data between tags is passed from the parser to the derived class by calling `self.handle_data()` with the data as argument. Entity references are passed by calling `self.handle_entityref`() with the entity reference as the argument. Numeric character references are passed to s`elf.handle_charref()` with the string containing the reference as the argument.

Lispy: Scheme Interpreter in Python (Lispy 2.0)
Author: Peter Norvig
Source code: http://norvig.com/lispy.py

Why Scheme?

Scheme is a functional programming language. So, Scheme programs are made of only of expressions which makes it syntactically pure.

This is way different from C++ or Java as they are Object Oriented Programming languages. We are building a compiler for COOL language for our course project which can be said as a subset of Java and it is an OOP language. So, we know how a parser and a lexer works for such imperative languages. So, I wanted to explore the parser of a functional programming language. That's the reason why I chose Scheme.

Why Lispy 2.0 was chosen?

First, I went through Lispy 1.0 which is a small minimalistic interpreter for Scheme. But it had a lot of shortcomings. It didn't support some basic data types like strings, characters, booleans,etc. And moreover it didn't have a proper error recovery routine. It doesn't support even comments. But such a minimalistic interpreter helped me understand the basics pretty well. Then I went on to read the more advanced version of Lispy (Peter Norvig calls it three times more complicated).

Lispy 2.0 covered most of the shortcomings of Lispy. This was the reason why I chose to analyse Lispy 2.0 as I felt it was a more complete Lisp Interpreter.

Basic structure of the Interpreter:

Peter Norvig basically divides his interpreter into two parts:

1. Parsing (including lexing)
2. Evaluation

He uses eval( ) function for the evaluation part. In this report we'll concentrate only on the parsing and lexing parts and we won't go into the details of evaluation.

Simple diagram for his interpreter:

program (str) ➡ parse ➡ abstract syntax tree (list) ➡ eval ➡ result (object)

How is the input read?

For reading input he has used an InputPort object which wraps a file object and it also keeps track of the last line of the text read.

Basically, our tokenizer will act on the last line of the input read. i.e. it tokenizes the last line of the input read. These input ports will be able to read expressions as well as raw characters i.e. he has defined separate functions to read characters and to a read a line given an Inputport.

Tokenizer of Lexer:

He had used a normal python tokenizer with space as a delimiter for lexer in Lispy 1.0. But now, we can't use it as a space can appear in between strings also. In Lispy 2.0 , a complex regular expression was used as a delimiter for the python tokenizer.

tokenizer = r'"\s*(,@|[('`,)]|"(?:[\\].|[^\\"])*"|;.*|[^\s('"`,;)]*)(.*)'"

The above statement tokenizes the input line.

Features of the new tokenizer:

1. All the characters after a semicolon(;) have been gathered and that particular token is ignored.
2. An expression can span over multiple lines. In the previous version, a single expression had to be written in a single line.

Lexical analysis:

So, basically this is how my lexer works:

It reads the input program line by line with the help of an input port and tokenizes those lines using the tokenizer mentioned above. As a result, my tokens would have neglected all the comments as that is taken care by the tokenizer.

Drawbacks: Lexer doesn't contribute to error recovery. Generally in other programming languages, the list of tokens are well defined and each token will have it own regular expression. At the end if a character matches with none of the token, it is reported as an error. But, here that is not the case. We just use a delimiter to tokenize the input string. After the lexing is done, we don't know about the type of tokens.

Symbol class:

Unique symbols like define, set!, if, lambda,etc. are defined in a separate symbol class. So, during parsing when I want to check for equality of a symbol (e.g if) and a token, I will use this symbol class.

Parsing:

Basic parsing strategy:

My tokens are passed from the lexer to the parser. So, basically my parser reads the tokens one by one. The read_from_tokens( ) function creates a new list for each expression it encounters. So, it is possible that a list may contains another list inside it as nested expressions are allowed in Scheme.

When the parser encounters a ' (' it creates a new list and all the tokens between ' (' and ') ' are put into a new list. When my parser encounters a ') ' without encountering a corresponding ' (' it throws an "unmatched)" error. This parsing strategy is more than sufficient as my Scheme program is basically a list of expressions. But now we'll discuss about the error recovery routine in the parser.

Detection of type of a token:

This is taken care by the atom(token) function. This function interprets the tokens in the same way that python interprets the given token. i.e. atom( ) function uses python routines for type detection.

Error detection by parser:

The basic idea is to report the error during the definition of an expression rather than during the evaluation of the expression. We check for the validity of an expression as soon it is defined. For each each expression, the parser creates a corresponding list as mentioned before.

That list is passed to an error detection function. That function first detects the type of the expression and for that particular type it checks whether the passed list(expression) is legal or not.

Error Reporting:

The parser is quite robust to various kinds of syntax errors that can be committed by the user and they have generated user defined error messages. This is in contrast to our COOL Parser written in ANTLR where ANTLR had default error reporting routines and we used those default routines to display errors.