令和２年度学士論文

HTML5 字句解析仕様の
自然言語処理による意味解析

東京工業大学 情報理工学院 数理・計算科学系

学籍番号 17B01064

五十嵐彩夏

指導教員 南出靖彦 教授

提出日 1月18日

概要

概要．概要．概要．概要．概要．がいよう

目次

第 1 章　命令の抽出　1

1.1　Tag 型から Command への変換　1

1.2　条件分岐文の処理　7

1.3　命令への変換の例　8

1.4　1 状態の形式的な定義　9

第 2 章　字句解析器の実装　11

2.1　実装の概観　11

2.2　インタプリタの実装の詳細　12

第 3 章　実装の評価　13

3.1　実装の評価　13

第 4 章　結論　15

参考文献　17

第 1 章

命令の抽出

??章において自然言語処理し, その出力を利用して得た Tag 型の情報を用いて, **??**章で定式化した形での命令の抽出を行う.

1.1 Tag 型から Command への変換

Tag 型の値の木構造の形に関してのパターンマッチをし, 形式的な命令へ変換する. Tag 型から Command の形式への変換を関数として表記する.

$\mathcal{T}_S$	:	Tag → List [Command]	//	文 (NodeType が S である Node) から, Command 型のリストへ変換する関数
$\mathcal{T}_B$	:	Tag → Bool	//	条件文 (NodeType が S である Node) から, Bool 型へ変換する関数
$\mathcal{T}_{VP}$	:	Tag → List [Command]	//	動詞句 (NodeType が VP である Node) から, Command 型のリストへ変換する関数
$\mathcal{D}$	:	Tag → List [Tag]	//	名詞句 (NodeType が NP である Node) から, 1 単位ごとの名詞句に分割する関数
$\mathcal{T}_{NP_C}$	:	Tag → CommandValue	//	名詞句 (NodeType が NP である Node) から, CommandValue 型へ変換する関数
$\mathcal{T}_{NP_I}$	:	Tag → InplementVariable	//	名詞句 (NodeType が NP である Node) から, ImplementVariable 型へ変換する関数

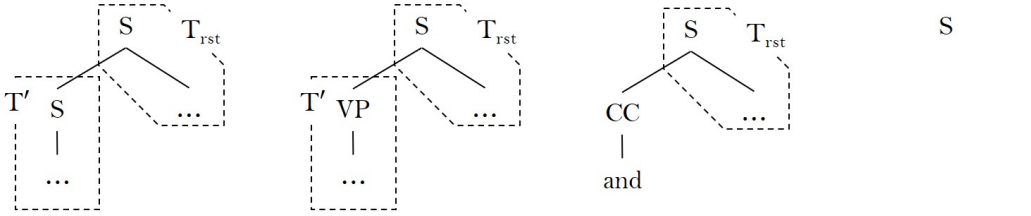
これらを順に説明していく. 木の形のマッチに関しては, 葉の値は単語の原型の情報のみ表記, 使用する.

1.1.1 文 (S ノード) の変換 $\mathcal{T}_S$

$\mathcal{T}_S$ は Tag 型の値 T を受け取り, T の形にマッチしたものに応じた Command 型のリストの値を返す.

文, 句の分解

複数の文, 句から構成されている時, Stag の子ノードを先頭から順に処理していく.
根の子ノードの先頭のノード形によって場合分け
Tag 型の値 T の形が,

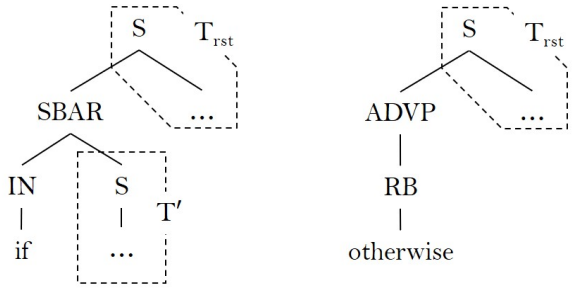


(i) 先頭が S　(ii) 先頭が VP　(iii) 先頭が繋ぎの単語　(iv) 子ノードが空

(i) の場合, $\mathcal{T}_S(\mathrm{T}')$ ++ $\mathcal{T}_S(\mathrm{T}_{rst})$ を返す.
(ii) の場合, $\mathcal{T}_{VP}(\mathrm{T}')$ ++ $\mathcal{T}_S(\mathrm{T}_{rst})$ を返す. (木 T′ に関して, 動詞句の変換を行う.)
(iii) の場合, $\mathcal{T}_S(\mathrm{T}_{rst})$ を返す. (“and” などの文を繋ぐ単語は無視する.)
(iv) の場合, [ ] を返す. (空の Commnad 型のリスト)

条件分岐文の場合

Tag 型の値 T の形が,

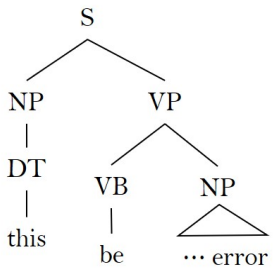


(i)　(ii)

(i) の場合, [If_($\mathcal{T}_B(\mathrm{T}')$)] ++ $\mathcal{T}_S(\mathrm{T}_{rst})$ を返す.
(ii) の場合, [Otherwise_()] ++ $\mathcal{T}_S$(Node(S,rst)) を返す.

Error 文のマッチ

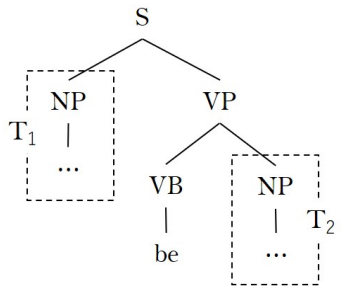
Tag 型の値 T の形が,



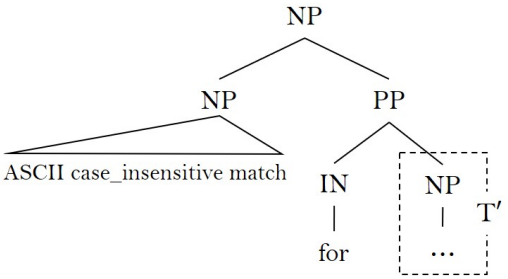
の時, [Error(··· error)] を返す.

1.1.2 条件文の変換 $\mathcal{T}_B$

$\mathcal{T}_B$ は Tag 型の値 T を受け取り, T の形にマッチしたものに応じた Bool 型の値を返す.



T_1 の限定詞を除いた語が “current end tag token”, T_2 の限定詞を除いた語が “appropriate end tag token” の時, CurrentEndTagIsAppropriate()
T_2 の形が,



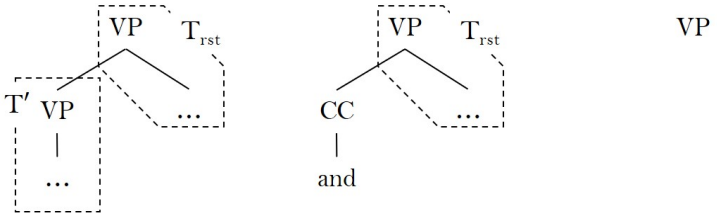
の場合, AsciiCaseInsensitiveMatch($\mathcal{T}_{NP_C}(T_1)$, $\mathcal{T}_{NP_C}(T'_2)$)
それ以外の時, IsEqual($\mathcal{T}_{NP_C}(T_1)$, $\mathcal{T}_{NP_C}(T_2)$)
また, T の文字列が, “character reference was consumed as part of an attribute” の時, CharacterReferenceConsumedAsAttributeVal()

1.1.3 動詞句 (VP ノード) の変換 $\mathcal{T}_{VP}$

$\mathcal{T}_{VP}$ は Tag 型の値 T を受け取り, T の形にマッチしたものに応じた Command 型のリストの値を返す.

動詞句の分解

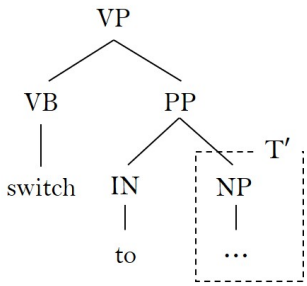
子ノードに動詞句を持つパターン
根の子ノードの先頭のノード形によって場合分け
Tag 型の値 T の形が,



(i) の場合, $\mathcal{T}_{VP}(\mathrm{T}') +\!+ \mathcal{T}_{VP}(\mathrm{T}_{rst})$ を返す.
(ii) の場合, $\mathcal{T}_{VP}(\mathrm{T}_{rst})$ を返す. (“and” などの文を繋ぐ単語は無視する.)
(iii) の場合, [ ] (空の Commnad 型のリスト) を返す.

Switch 文のマッチ

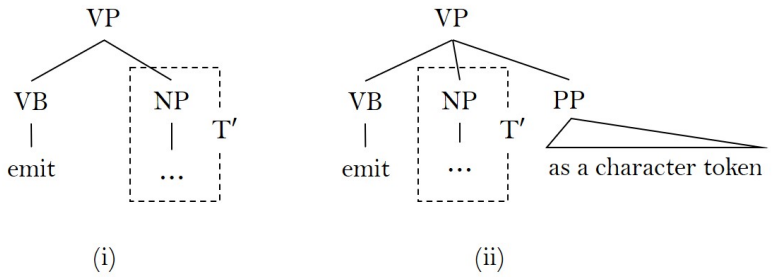
命令文の構文木解析の形を調べ, それに基づいた Tag 型の値のパターンマッチを作る. Tag の形が,



の時, [Switch($\mathcal{T}_{NP_C}(\mathrm{T}')$)] を返す.

Emit 文のマッチ

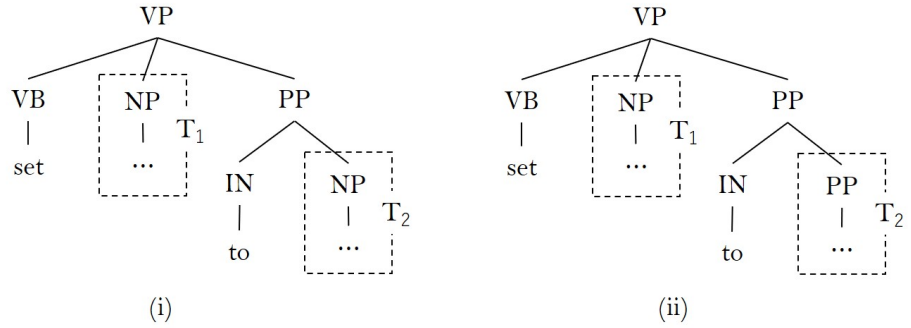
木構造が,



の場合, $\mathcal{D}(\mathrm{T}') = [\mathrm{T}'_1, \ldots, \mathrm{T}'_n]$ の時, [Emit($\mathcal{T}_{NP}(\mathrm{T}'_1)$), …,Emit($\mathcal{T}_{NP}(\mathrm{T}'_n)$)] を返す.
$\mathcal{D}(\mathrm{T}') = [\mathrm{T}'_1, \ldots, \mathrm{T}'_n]$ の時,
[Emit(CharacterToken($\mathcal{T}_{NP}(\mathrm{T}'_1)$)), …,Emit(CharacterToken($\mathcal{T}_{NP}(\mathrm{T}'_n)$))] を返す.

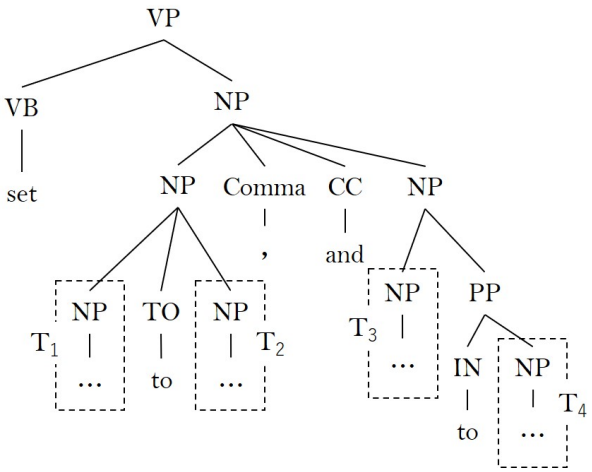
Set 文のマッチ

木構造が,



の場合, [Set($\mathcal{T}_{NP_I}$(Node(NP, np1)), $\mathcal{T}_{NP_C}$(Node(NP, np2)))] を返す.
の場合,
pp が “on” だったら, [Set($\mathcal{T}_{NP}$(Node(NP, np)), On)] を返す.

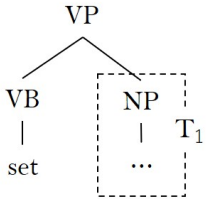
■個別に対応した Set 文のマッチ “Set that attribute’s name to the current input character, and its value to the empty string.” みたいな文は木構造が上記のものと違うものになってきてしまうので, これを個別に対応した.
木構造が,



のとき,

[Set($\mathcal{T}_{NP}$(T$_1$), $\mathcal{T}_{NP}$(T$_2$)), Set($\mathcal{T}_{NP}$(T$_3$), $\mathcal{T}_{NP}$(T$_4$))] を返す.

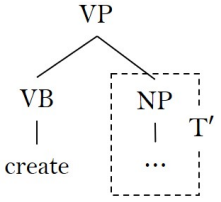
“Set the self-closing flag of the current tag token.” のような状態の切り替え先を示す “to on” が省略されている場合がある. このような文の構文木は



となる. よってこの形にマッチした場合は, [Set($\mathcal{T}_{NP}$(Node(NP, np)), On)] を返す.

Create 文のマッチ

木構造が,



の場合

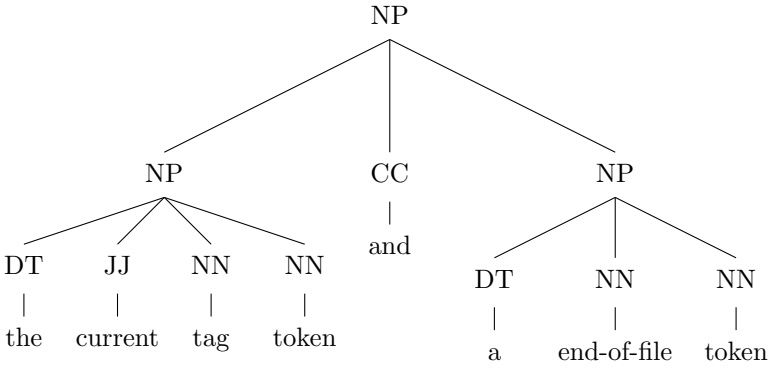
T′ の単語に番号 n の参照関係が存在 ⇒ [Create(“x$_n$”, $\mathcal{T}_{NP}$(T′))] を返す.

そうでない ⇒ [Create(“”, $\mathcal{T}_{NP}$(T′))] を返す.

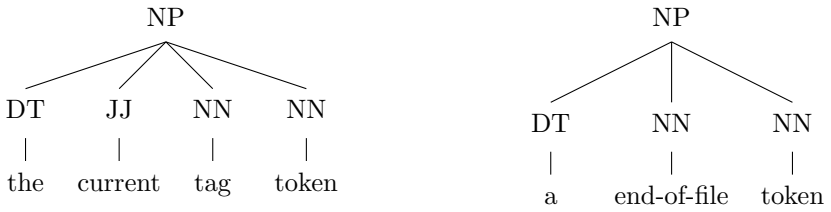
1.1.4　名詞句 (NP ノード) の分割 $\mathcal{D}$

$\mathcal{D}$ は Tag 型の値 T を受け取り, Tag 型のリストの値を返す.
NP ノードを受け取って, NP ノードのリストを返す関数.
“A and B” や “A, B” など 1 つの名詞句の中に複数の名詞が含まれている場合もある. それらを分解し, 名詞のリストとして出力する.

例えば, “the current tag token and an end-of-file token”



これを,



という風にする.

1.1.5　名詞句 (NP ノード) の変換 $\mathcal{T}_{NP_C}$, $\mathcal{T}_{NP_I}$

$\mathcal{T}_{NP_C}$, $\mathcal{T}_{NP_I}$ は Tag 型の値 T を受け取り, それぞれ CommandValue 型, ImplementVariable 型の値を返す.
HTML5 の仕様書には様々な変数の言葉や値の言葉が出てくる. NP ノード内の自然言語で記述している言葉から, CommandValue 型や ImplementVariable 型というプログラムが認識できる形にする際, 単純に文字列に変数名や値などの特定の単語が含まれているかどうかを調べるというやり方をとった.

例えば, 変換対象の NP ノード内に”return state”という言葉が含まれていた場合は, ReturnState を返す. 特定の単語にマッチせず, かつ参照番号が-1 でない数字 n のとき (その単語にラベル n の参照関係があるとき) は 変数”x_n”という意味を持つ, Variable(”x_n”) を返す. name, value が含まれている場合は NameOf(Variable(”x_n”)), ValueOf(Variable(”x_n”))

1.2　条件分岐文の処理

Command 変換の時に, If 文をまとめてあれしなかったのは, 自ら組み立てようとする方針にしたのは, Otherwise の処理がどこまでなのかの判別が難しかったからである. 仕様書内の条件分岐の文 (If 文) に関し

て, Command 型のリストに変換する際, 条件分の範囲を判断するときに手を加えた.
例えば, 仕様書内の
“If ···, then ···. Otherwise, switch to the script data escaped state. Emit the current input character as a character token.” という文の, “Emit the current input character as a character token.” の部分が Otherwise の中の文として処理したいのか曖昧である. このような文は人でも判断しずらいものだから, 機械による処理でも正しく解釈するのは難しい. よって条件分岐文に関しては, 手動でどこまでが otherwise にの処理に入る文なのかを判断するようにした.

具体的な処理としては, Tag 型からの変換によって出力された Command のリストに以下のような形があったら,

If_(b)
command_1
···
command_m
Otherwise_()
command_{m+1}
···
command_n

これを考えられる If 文に組み立て, そこから正しいほうを人の手で選ぶ.

例えば, 条件文の処理をする前の Command 列
[If_(b), command_1, Otherwise_(), command_2, command_3] は,
⇒ [If(b, [command_1], [command_2, command_3])]
⇒ [If(b, [command_1], [command_2]), command_3]
と 2 つ出力され, 正しい解釈の方を手動で選び, 取り出す命令を決定する.

1.3 命令への変換の例

例**??**の “Create a comment token. Emit the token.” から命令型 Command のリストを抽出する.
“Create a comment token.” から得られた木を T_1, “Emit the token.” から得た木を T_2 と置く. (ROOT タグは省略) また, T_1, T_2 の部分木を以下の図 1.1 のように置く. (点線部分)

T_1, T_2 をそれぞれ $\mathcal{T}_S$ に適用すると,

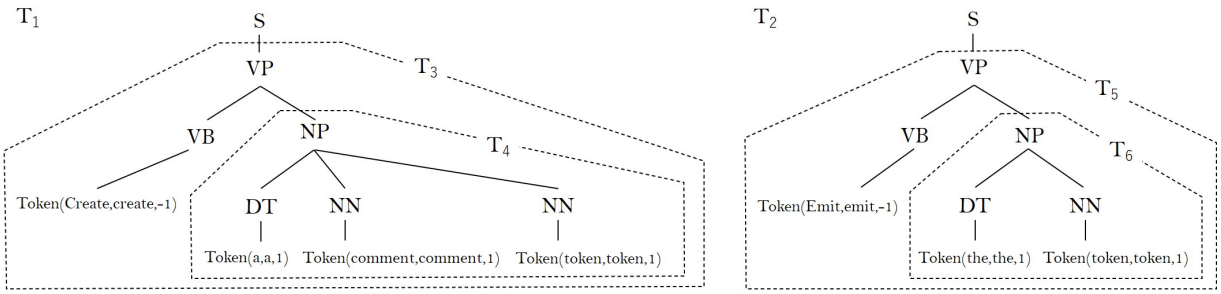


図 1.1　例**??**で変換された Tag 型

$$
\begin{aligned}
\mathcal{T}_S(\mathrm{T}_1) &= \mathcal{T}_{VP}(\mathrm{T}_3) +\!+ \mathcal{T}_S(\mathrm{Node(S, Nil)}) \\
&= \mathcal{T}_{VP}(\mathrm{T}_3) +\!+ [\,] \\
&= [\ \mathrm{Create}(\mathcal{T}_{NP}(\mathrm{T}_4))\ ] && \text{// } \mathrm{T}_3\text{は Create 文にマッチする} \\
&= [\ \mathrm{Create}(\text{“x\_1”}, \mathrm{NewCommentToken})\ ] && \text{// } \mathrm{T}_4\text{がもつ文字列に “comment token” が含まれている.} \\
& && \quad \text{その文字列が, 番号が 1 の参照関係を持っている.}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_S(\mathrm{T}_2) &= \mathcal{T}_{VP}(\mathrm{T}_5) +\!+ \mathcal{T}_S(\mathrm{Node(S, Nil)}) \\
&= \mathcal{T}_{VP}(\mathrm{T}_5) +\!+ [\,] \\
&= [\ \mathrm{Emit}(\mathcal{T}_{NP}(\mathrm{T}_6))\ ] && \text{// } \mathrm{T}_5\text{は Emit 文にマッチする} \\
&= [\ \mathrm{Emit}(\mathrm{Variable}(\text{“x\_1”}))\ ] && \text{// } \mathrm{T}_6\text{がもつ文字列 “the token” は番号が 1 の参照関係を持っている}
\end{aligned}
$$

となり, Command 型のリスト
[Create(Variable(“x_1”), NewCommentToken) , Emit(Variable(“x_1”))] が得られる.

1.4　1 状態の形式的な定義

実装では, 1 状態あたりの形式化した内容を記述するクラスとして `StateDef` を定義する. `StateDef` は状態名, 文字マッチ前の処理, 文字マッチの処理を有する.

```
case class StateDef(
  stateName:  String,                       // 状態名
  prevProcess :  List[Command],             // 文字マッチ前の処理
  trans :  List[(String, List[Command])]    // 文字マッチ（文字 , その文字の処理）のリスト
)
```

1.4.1　例

表 1.1 の状態に**??**, 1 章の処理をし, 1 状態の形式的な定義に変換すると, 図 1.1 のようになる.

表 1.1　HTML5 字句解析仕様の状態：RCDATA less-than sign state

状態名	RCDATA less-than sign state	
文字マッチ前の処理	Consume the next input character:	
文字マッチ処理	文字	処理
	U+002F SOLIDUS (/)	Set the temporary buffer to the empty string. Switch to the RCDATA end tag open state.
	Anything else	Emit a U+003C LESS-THAN SIGN character token. Reconsume in the RCDATA state.

Listing 1.1　RCDATA less-than sign state の形式化

```
// 状態名
stateName = RCDATA_less_than_sign_state
// 文字マッチ前の処理
prevProcess = [ Consume(NextInputCharacter) ]
// 文字マッチ
trans = [
   ( U+002F SOLIDUS (/),
         [ Set(ITemporaryBuffer, CString()), Switch(RCDATA_end_tag_open_state) ] ),
   ( Anything else,
         [ Emit(CString(<)), Recomsume(RCDATA_state) ] )
]
```

第 2 章

字句解析器の実装

1 章で形式化した命令を「字句解析の状態の定義」として入力にとる, HTML5 の字句解析をするインタプリタを実装した. 実装はプログラミング言語 Scala で行った.

2.1　実装の概観

字句解析インタプリタは, 図 2.1 のような感じで, 入力として 字句解析器の処理の定義, 字句解析器の環境をとり, 字句解析トークン列, 構文エラー, 新しい字句解析器の環境 を返す.

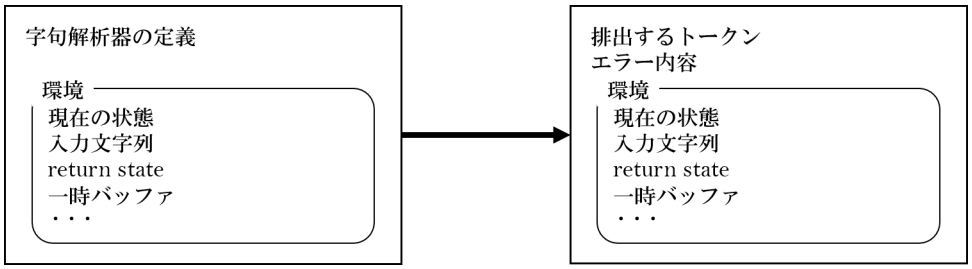


図 2.1　インタープリタ概略

字句解析インタプリタの処理の流れは, まず, 字句解析器の環境の “現在の状態” を参照し, “字句解析器の処理の定義” から現在の状態に対応する処理を取り出す. 文字マッチ前の処理の部分に入っている Command 型の列を取り出し, それを前から 1 つずつ Command 型の解釈の関数によって実行していく.

次に, 字句解析器の環境の “現在消費した文字” を参照し, その文字に関して文字の種類のマッチングを行い, その文字にマッチする処理 (Command 型の列) を取り出す.

そして, 取り出した Command 型の列をそれを前から 1 つずつ Command 型の解釈の関数によって実行していく.

この一連の処理を, 字句解析インタプリタが EOF トークンを排出するまで繰り返す.

2.1.1　字句解析器の環境

字句解析仕様に出てくる語句を実装数字句解析器の環境の変数としてとる. 環境の変数には, 現在の状態, 入力文字列から消費したばかりの文字を表す “現在消費した文字”, 状態名を値として持つ “return state” 等がある.

2.2 インタプリタの実装の詳細

2.2.1 字句解析トークン

字句解析器によって出力されるトークンは, Token というトレイトとして実装した. Token は DOCTYPE トークンを表す DOCTYPEToken(s,s,s,b), 開始タグトークンを表す startTagToken(), 終了タグトークンを表す endTagToken(), コメントトークンを表す commentToken(s), 文字トークンを表す characterToken(s), EOF トークンを表す endOfFileToken() を有する.

2.2.2 Value 型

インタープリタで扱う変数の値の型として Value をトレイトとして実装した. Value は Int の値を表す IntVal(int: Int), Boolean の値を表す BooleanVal(boolean: Boolean), Char の値を表す CharVal(c: Char), String の値を表す StringVal(string: String), 文章の終端 EOF を表す EOFVal, 字句解析器の状態名を表す StateVal(statename: String), 字句解析トークンの値を表す TokenVal(token: Token) を有する.

2.2.3 文字マッチングの処理

字句解析器の現在の状態の文字マッチの処理の定義と現在消費した文字 (current inputCharacter) を受け取り, マッチした文字に対応する処理 Command 型のリストを返す処理を実装した.

2.2.4 CommandValue 型の解釈

CommandValue 型と環境を受け取り, Value 型を返す関数を実装した.

CommandValue 型の解釈例：ReturnState

字句解析器の環境の変数である, Value 型の値 returnState を返す.

2.2.5 Command 型の解釈

Command 型と環境を受け取り, 新しい環境, 排出トークン, エラー内容を返す関数を実装した. Command 型の値ごとにその処理を記述した.

Command 型の解釈例：Switch 文

字句解析器の環境の変数である, “現在の状態” に, CommandValue 型の引数を CommandValue 型の解釈する関数で Value 型にした値を代入する.

第 3 章

実装の評価

3.1　実装の評価

2 章で実装したインタプリタの関数をそれぞれテストした後, html5lib-tests [1] の tokenizer のテストデータを用い, 抽出した命令を定義として入れた, 字句解析インタープリタのテストをし, 抽出した命令の正しさを検証した.

3.1.1　HTML5 字句解析テスト

使用した字句解析器のテストデータは, 1 つのテストファイルに複数のテストが含まれており, 1 つのテストあたり, 入力文字列, 出力するトークン列, 字句解析する際の初期状態, 直前に排出されている開始タグトークン名, 出力されるエラーの情報を持つ.

テストは, テストデータの入力文字列, 初期状態, 直前の開始タグトークン名を字句解析インタプリタの初期の環境として設定し, 字句解析を行い, テストデータの字句解析トークンの出力と, 実装したインタプリタによる字句解析トークンの出力を比べる. また, テストデータのエラーの出力と, 実装したインタプリタによるエラーの出力も比較する. そしてそれらが一致していたら成功とする.

思ったこと

Tag 型からの命令抽出に関しては, 特殊な部分を逐一個別に対応していたので上手くいったと思われる.

domjs が上手くいかなかった原因

CDATA の部分の処理を実装していなかったため.

3.1.2　問題点

“If the six characters starting from the current input character are an ASCII case-insensitive match for the word “PUBLIC”, then consume those characters” この文章を自然言語解析させると “those characters” は “the six characters starting from the current input character” を参照するという出力になる.
もし, この状態へ遷移した時点での入力文字列が “public ···” であったら, まず文字’p’ を消費し, 入力文字列が “ublic ···” となる.
機械的にこの文章を処理しようとすると, 現在の入力文字列 “ublic ···” から文字列 “public” を消費せよとい

テストファイル名	結果	テスト内容
contentModelFlags.test	24/24	
domjs.test	57/58	
entities.test	80/80	&から始まる文字列の文字の参照が上手くいっているか
escapeFlag.test	9/9	偽のコメントトークンに対する処理
namedEntities.test	4210/4210	named character references の表の参照が上手くいっているか
numericEntities.test	336/336	character reference code から文字への参照が上手くいっているか
pendingSpecChanges.test	1/1	コメントトークン中に EOF トークンが出てきた場合のテスト
test1.test	68/68	テスト 1
test2.test	35/45	テスト 2
test3.test	1374/1786	テスト 3
test4.test	81/85	テスト 4
unicodeChars.test	323/323	ユニコード表記の文字列が対応する文字に変換されているか
unicodeCharsProblem.test	5/5	不適切な場合のユニコードの処理が上手くいっているか

う解釈になるので, 上手くいかない.
この問題を手動で解決させた結果, 以下のようなテスト結果の改善が成された.

テストファイル名	結果
domjs.test	57/58
test2.test	45/45
test3.test	1786/1786
test4.test	85/85

第 4 章

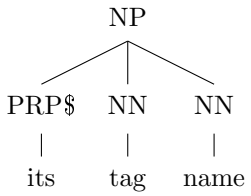
結論

本論文では, HTML5 の字句解析仕様に自然言語処理を適用させ, 命令の抽出が行えることを確認した.

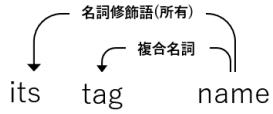
本研究で扱った仕様に関しては, 命令の種類や記述の仕方が限られており, 構文木の情報から命令の抽出が可能だった. しかし, 構文木解析の情報のみではなく係り受け解析の情報も利用したほうがいいと思われる部分もあった.

例えば, Tag 型から命令の型である Command 型へ変換する際, 木構造のマッチングにおいて, Emit 文のマッチの部分で, “Emit ⟨Emit の対象 ⟩” のような単純な文ではなく, “Emit the current input character as a character token.” のように “as a character token” という補足的な情報が加わると, 構文木の形が変わるので, 複数種類のパターンマッチをする必要が出てくる. その点において, 係り受け解析から得られる情報だとEmit する対象となる文字を直接知ることが出来るので, 係り受け解析を用いたほうが簡潔に命令を抽出できる場合があると感じた.

また, NP ノードから CommandValue 型への変換する際に関しても, 例えば, “its tag name” という名詞句を例にとると,
この文の構文木解析は,



という結果だが, 係り受け解析だと,



という結果が得られ, “name” が “its” のものであるということがはっきりわかり, 得られる情報が多い.
よって, NP ノードを解析する際, 構文木解析と係り受け解析併用したほうがいいと思った.

謝辞

謝辞. 謝辞. 謝辞. 謝辞.

参考文献

[1] James Graham, Geoffrey Sneddon, et al. html5lib-tests, 2020. https://github.com/html5lib/html5lib-tests.