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

序論

自然言語は、人間が同士が互いにコミュニケーションをとるために発展してきた言語である. そして自然言語をコンピュータにで処理する技術を自然言語処理（Natural Language Processing）と呼んでいる. 本論文では自然言語処理の技術を使って HTML5 の字句解析仕様から命令を抽出することを試みた.

図 1.1 が HTML5 の字句解析仕様の意味解析の概要である.

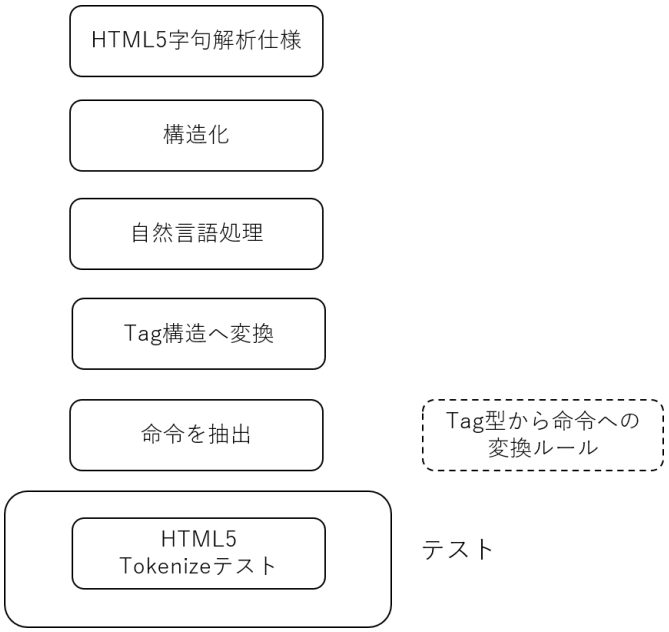


図 1.1　流れ（仮）

本論文では, まず 2 章で自然言語処理の基礎知識を述べる. 次に 3 章で HTML5 の字句解析器の主な仕様, 動作について述べる. 4 章で抽出する命令の形式を BNF として述べ, 5 章で自然言語処理ライブラリを用い, それを HTML5 字句解析仕様に適用させ, 6 章で自然言語処理の出力をもとに仕様書の命令の抽出を行った. 7 章で抽出した命令をもとに字句解析をするインタプリターを作成し, 8 章で字句解析のテストデータを用い, 抽出した命令の正しさを検証した.

第 2 章

準備

2.1　品詞タグ

準備 [3] S : 節 VP : 動詞句 NP : 名詞句 VB : 動詞

2.2　自然言語処理

2.2.1　使用ライブラリ

自然言語処理のライブラリとして,Stanford CoreNLP [2] を使用した. Stanford CoreNLP は自然言語処理ツールのひとつであり, スタンフォード大学によって提供されている. StanfordCoreNLP では, 形態素解析 (品詞タグ付け、単語の原型の取得), 構文解析, 意味解析などが出来る.
(pipeline の説明をする)

2.2.2　概要

”Mika likes her dog’s name.” を Stanford CoreNLP で自然言語処理をさせる.

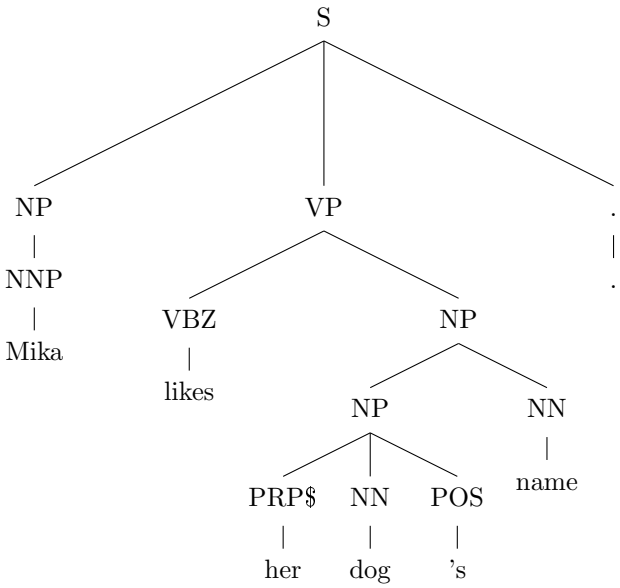
トークン分割、品詞タグ付け、原型

Mika/NNP likes/VBZ her/PRP$ dog/NN ’s/POS name/NN ./.
Mika/Mika likes/like her/she dog/dog ’s/’s name/name ./.

固有表現抽出

数字や時間、アドレス、人間、地名といった固有な表現を抽出することが出来る.
Mika : PERSON

構文木解析



係り受け解析

係り受け解析とは、単語間の関係を解析するものである.

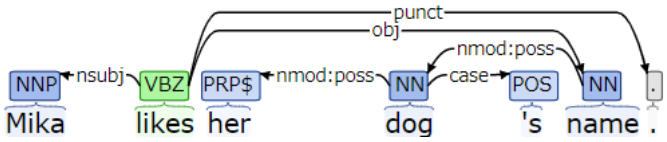


図 2.1　dependency

参照関係の解析

参照関係の解析とは, 文章内で複数個同じものを指し示す単語がある時、それを抽出するものである.it や he などの指示語の指し示すものを見つける時になどに使用される.

Mika, her

第 3 章

HTML5 字句解析仕様

3.1　概要

token

HTML5 字句解析器は HTML5 の文章を token という単位に分解する. 字句解析器により排出されたトークンは DOM ツリーを構成する次のステップに使われる. 排出される token には,5 つ種類があり, その文書で利用する HTML や XHTML のバージョンを表す DOCTYPE token, ”<!– –>”などでコメントアウトした文章を表す comment token, <,> で囲まれているタグを表す start tag token と end tag token, 文字を表す character token, 文章の終了を表す end-of-file token がある.

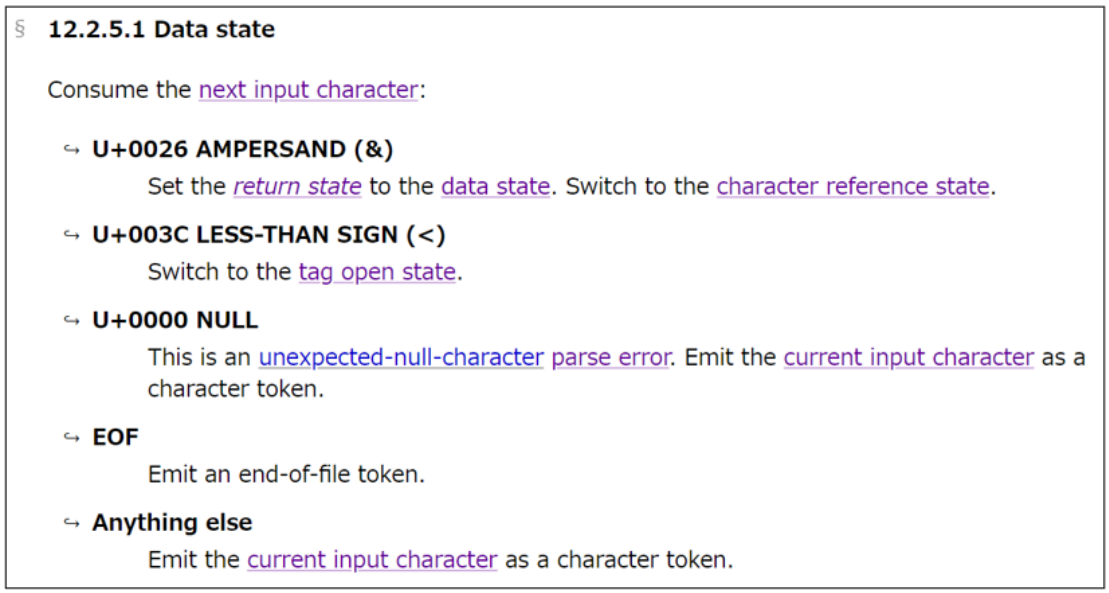
DOCTYPE token は DOCTYPE の名前, public identifier と system identifier の値, force-quirks flag の要素を持っている.
start tag token, end tag token はタグの名前, タグの属性の集合 (attributes), self-closing flag の要素を持っている.
end-of-file トークンが排出されたら字句解析器の動作は終了する.

変数

HTML5 字句解析器は return state や一時バッファなどの変数を持つ.

状態

HTML5 字句解析仕様は WHATWG community の web サイトから得られる. [4]
HTML5 の字句解析器は 80 個の状態のあるオートマトンとして定義されている. それぞれの状態は以下の図 3.1 の形式で書かれている.



§ **12.2.5.1 Data state**

Consume the next input character:

↪ **U+0026 AMPERSAND (&)**
Set the *return state* to the data state. Switch to the character reference state.

↪ **U+003C LESS-THAN SIGN (<)**
Switch to the tag open state.

↪ **U+0000 NULL**
This is an unexpected-null-character parse error. Emit the current input character as a character token.

↪ **EOF**
Emit an end-of-file token.

↪ **Anything else**
Emit the current input character as a character token.

図 3.1　HTML5 字句解析仕様書

3.2　動作の例

3.2.1　例 1

入力”<a>bc”に対して, HTML5 字句解析器は以下のように動作を行う.

1. 初期状態 Data state から, 文字’<’ が消費され, Tag open state に遷移する.
2. 文字’a’ を消費し, 名前が空文字である新たな start tag token を作る. Tag name state に遷移する.
3. 先ほど消費した文字’a’ を再度消費し, start tag token の名前に’a’ を付け足す.
4. 文字’>’ を消費し, start tag token を排出し, Data state に遷移する.
5. 文字’b’ を消費し, characterToken(’b’) を排出する.
6. 文字’c’ を消費し, characterToken(’c’) を排出する.
7. 文字’<’ を消費し, Tag open state に遷移する.
8. 文字’/’ を消費し, End tag open state に遷移する.
9. 文字’a’ を消費し, 名前が空文字である新たな end tag token を作る. Tag name state に遷移する.
10. 先ほど消費した文字’a’ を再度消費し, end tag token の名前に’a’ を付け足す.
11. 文字’>’ を消費し, end tag token を排出し, Data state に遷移する.
12. end-of-file token を排出する.

動作の結果として,
startTagToken(name = ”a”, attributes = []), characterToken(’b’), characterToken(’c’), endTagToken(name = ”a”), end-of-fileToken
が順に排出される.

3.2.2 例 2

入力”a<ab”に対して, HTML5 字句解析器は以下ように動作を行う.

1. 初期状態 Data state において, 文字’a’ を消費し, characterToken(’a’) を排出する.
2. 文字’<’ を消費し, Tag open state に遷移する.
3. 文字’a’ を消費し, 名前が空文字である新たな start tag token を作る. Tag name state に遷移する.
4. 先ほど消費した文字’a’ を再度消費し, start tag token の名前に’a’ を付け足す.
5. 文字’b’ を消費し, start tag token の名前に’b’ を付け足す.
6. ”eof-in-tag”構文エラーを出す. end-of-file token を排出する.

動作の結果として,
characterToken(’a’), end-of-fileToken が順に排出される.

第 4 章

抽出の形式

4.1　抽出する命令の形式

以下の BNF の形式で仕様書から抽出する命令を定義する.

cList : CommandList ··· 命令文のリスト

c : Command ··· 命令文

b : Bool ··· 条件文

cval : CommandValue ··· 値

ival : InplementVariable ··· 代入される変数

cList ::= c :: cList | Nil

c ::= If(b, $cList_1$, $cList_2$) // if b then $cList_1$ else $cList_2$

| Ignore() // 何もしない

| Switch(cval) // 状態 cval へ遷移する

| Reconsume(cval) // 状態 cval へ遷移. この状態で消費した文字を, 次の状態で再度消費する.

| Set(ival, cval) // ival に cval を代入する (ival ← cval)

| AppendTo(cval, ival) // ival に cval を追加する (ival ← ival + cval)

| Emit(cval) // トークン cval を排出する

| Create(cval) // トークン cval を新たに作る

| Consume(cval) // 文字 cval を消費する

| Error(string) // エラー string を排出する

| FlushCodePoint() // 一時バッファの内容を排出する

| StartAttribute() // 現在の tagToken に新しい属性を加える

| TreatAsAnythingElse() // AnythingElse の処理内容を実行する

| AddTo(cval, ival) //ival ← ival + cval

| MultiplyBy(ival, cval) //ival ← ival * cval

b ::= And(b_1, b_2)
| Or(b_1, b_2)
| Not(b)
| CharacterReferenceConsumedAsAttributeVal() // CharacterReferenceCode が属性の値として消費されているか
| CurrentEndTagIsAppropriate() //EndTagToken が適切であるか
| IsEqual($cval_1$, $cval_2$)

cval ::= StateName(string) // 状態名 string
| ReturnState // return state
| TemporaryBuffer // temporary buffer
| CharacterReferenceCode // character reference code
| NewStartTagToken // 新しい start tag token
| NewEndTagToken // 新しい end tag token
| NewDOCTYPEToken // 新しい DOCTYPE token
| NewCommentToken // 新しい comment token
| CurrentTagToken // 一番新しく作られた tag token
| CurrentDOCTYPEToken // 一番新しく作られた DOCTYPE token
| CurrentAttribute // 一番新しく作られた attribute
| CommentToken // 一番新しく作られた comment token
| EndOfFileToken // end of file トークン
| CharacterToken(char) // character token : char
| LowerCase(cval) // cval の小文字
| NumericVersion(cval) // 16 進数表記されている cval の数字としての値
| CurrentInputCharacter // 現在消費した文字
| NextInputCharacter // 入力文字列の一番最初の文字
| Variable(string) // 変数 string
| CChar(char) // Char 型の値 char
| CString(string) // String 型の値 string
| CInt(int) // Int 型の値 int
| CBool(boolean) // Boolean 型の値 boolean

ival ::= IReturnState // return state
| ITemporaryBuffer // 一時バッファ
| ICharacterReferenceCode // character reference code
| ICurrentTagToken // 一番新しく作られた tag token
| ICurrentDOCTYPEToken // 一番新しく作られた DOCTYPE token
| ICurrentAttribute // 一番新しく作られた attribute
| ICommentToken // 一番新しく作られた comment token
| IVariable(string) // 変数 string
| INameOf(ival) // ival の名前
| IValueOf(ival) // ival の値
| IFlagOf(ival) // ival の flag
| SystemIdentifierOf(ival) // ival の system identifier
| PublicIdentifierOf(ival) // ival の public identifier

string,char,int,boolean はそれぞれ Scala の標準の型 (String,Char,Int,Boolean) の値

4.2 例

Switch to the Data_state.
⇒ Switch(StateName(Data_state))

Append the lowercase version of the current input character to the current tag token's tag name.
⇒ Append(LowerCase(CurrentInputCharacter), INameOf(CurrentTagToken))

第 5 章

自然言語処理

5.1　自然言語処理の対象

HTML5 の字句解析仕様には 80 個の状態があるが, 本論文では字句解析仕様の自然言語処理する対象は 80 個のうち 77 個とした.
なぜなら 80 個の状態のうち, 77 の状態は同じような構造で書かれているが, 残りの 3 状態 (Markup declaration open state, Named character reference state, Numeric character reference end state) はそれぞれ特殊な構造で書かれている. これらも一括りにして自然言語処理を適用させるのは複雑になると判断し, 自然言語処理の対象から除外した.
また, HTML5 仕様書内の Note や Example 等の補足説明は無視する.
尚, テストする際は残りの 3 つは手動で実装することにした。

5.2　対象の前処理

仕様書自体が構造的に書かれているので, 仕様書解析の入力はその HTML のソースコードとした.

5.2.1　Scala 構造体

HTML5 字句解析仕様書は構造的に書かれている.
状態名は”h5”タグ内にある.
文字マッチング前の処理は”h5”タグ直後の”p”タグ内に記述してある.
文字マッチングの処理は”dt”,”dd”タグ.

StateStructure

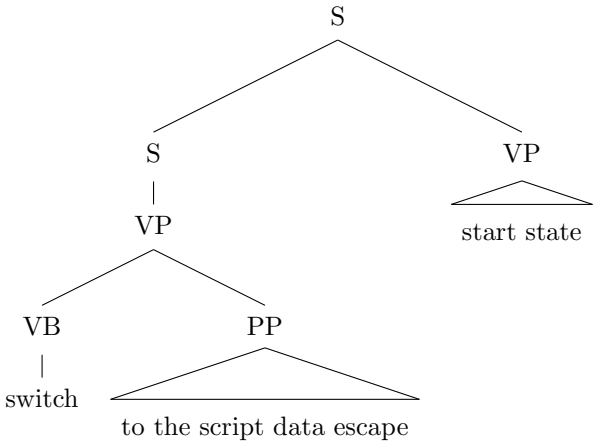
StateStructure 構造体は, 状態名 name, 最初の処理 prev, 文字マッチングの処理 trans を持つ. 仕様書の HTML ソースファイルの情報をこの構造にする.

5.2.2　文字列の置き換え

自然言語処理したい文章をそのまま処理すると, トークンの分割や品詞解析が適切な形で解釈されない.
よって自然言語処理する際に, 前処理として以下の文字列の置き換えをすることによって適切に文章が解釈されるようにした.

状態名の置き換え

Switch to the script data escape start state. の命令文が構文木解析において



と”script data escape”と”start state”が本来同じまとまりの中にいるべき単語がそれぞれ別のまとまりにいると解釈される.
よって状態名を 1 つのトークンとして扱われるようにし, 適切に命令の文が解釈されるようにするため, 次のような記法に置き換えることにした.
・空白､”-”を”_”にする.
・”(“,”)”を除く.
・先頭を大文字にする.
例:
attribute value (double-quoted) state ⇒ Attribute_value_double_quoted_state

Unicode の置き換え

仕様書内では”U+xxxx”というユニコードが多用されている. これにそのまま自然言語処理を行うと, 単語分割において”U”, ”+xxxx”と 2 つのトークンに分割される. よってユニコード内の”+”を”_”に置き換えることによって 1 つのトークンとして認識させるようにした.
例:
“U+00AB” ⇒ “U_00AB”

動詞の置き換え

自然言語処理の結果を確認してみると, 品詞解析の時点で動詞と認識されるべき単語が名詞扱いされることがあった.
例えば,"Reconsume"は"re"と"consume"の複合語であり, 一般的な辞書にも載ってないので動詞として解釈されないことがあった. よってこのような単語の前に"you"という単語を付け加え,"you Reconsume ···"とすることによって,"Reconsume"を動詞として解釈させるようにした.
また,Stanford CoreNLP は命令文の解釈が苦手である. よって特定の単語 (Switch, Reconsume, Emit, Flush, Append, Add, Multiply) の前に"you"という仮の主語を付け加え, 命令文にならないようにする.
例:
“Switch to the data state.” ⇒ “you Switch to the data state.”

その他の置き換え

- "-"で繋がれている単語は 1 つのトークンとして認識されないため,“-” を “_”に置き換えた.
- 句読点をまたいでいる場合, 参照関係の解析が上手くいかないことがあった. 参照関係が多く出てくる Set 文に関して, "(,|.) set" ⇒ “ and set” と置き換えをした.
- “!”が文末記号と認識されるため, “!” は “EXC” に置き換える.

5.3 Tag 型への変換

StanfordCoreNLP を用いての自然言語処理から得られる情報のうち, 単語の原型の情報, 構文解析の結果, 参照関係の解析の結果を使用した. そしてそれらの情報を,Scala で定義した Tag 型の構造に変換した.

5.3.1 Tag 型

Tag 型は,Node 型と Leaf 型の 2 種類を持っている. Node 型は構文木の句を表すもので, 句の種類を表す NodeType と, そのノードの子である Tag 型のリストを持つ. Leaf 型は構文木の末端である単語を表すもので, 品詞名を表す LeafType と, 単語の情報を格納する Token 型を持つ. Token 型は単語, 単語の原型, 参照関係の番号の情報を持つ.

Listing 5.1 Tag の定義

```
trait Tag
case class Node(node: NodeType, list: List[Tag]) extends Tag
case class Leaf(leaf: LeafType, token: Token) extends Tag
case class Token(word: String, lemma: String, coref: Int) extends Tag
trait NodeType
case object S extends NodeType
case object NP extends NodeType
case object VP extends NodeType
...
trait LeafType
case object NN extends LeafType
```

```
case object NNP extends LeafType
case object VB extends LeafType
...
```

5.3.2 Tag 型への変換

変換の対象として,"Create a token. Emit the token."を例にとる.

構文木の処理

基本的には自然言語処理の構文木の出力の形を保った状態で木構造である Tag 型に変換するが, 例外的に以下の処理を加える.

1．-NP-PRP-"you"となっている部分を取り除く.
2．PRN ノード,"("と")"の間にあるノードを取り除く.
3．ドット (.) を取り除く.
4．動詞を表す品詞は複数 (VB,VBZ,VBP...) あるが, それらは"VB"に統一する.

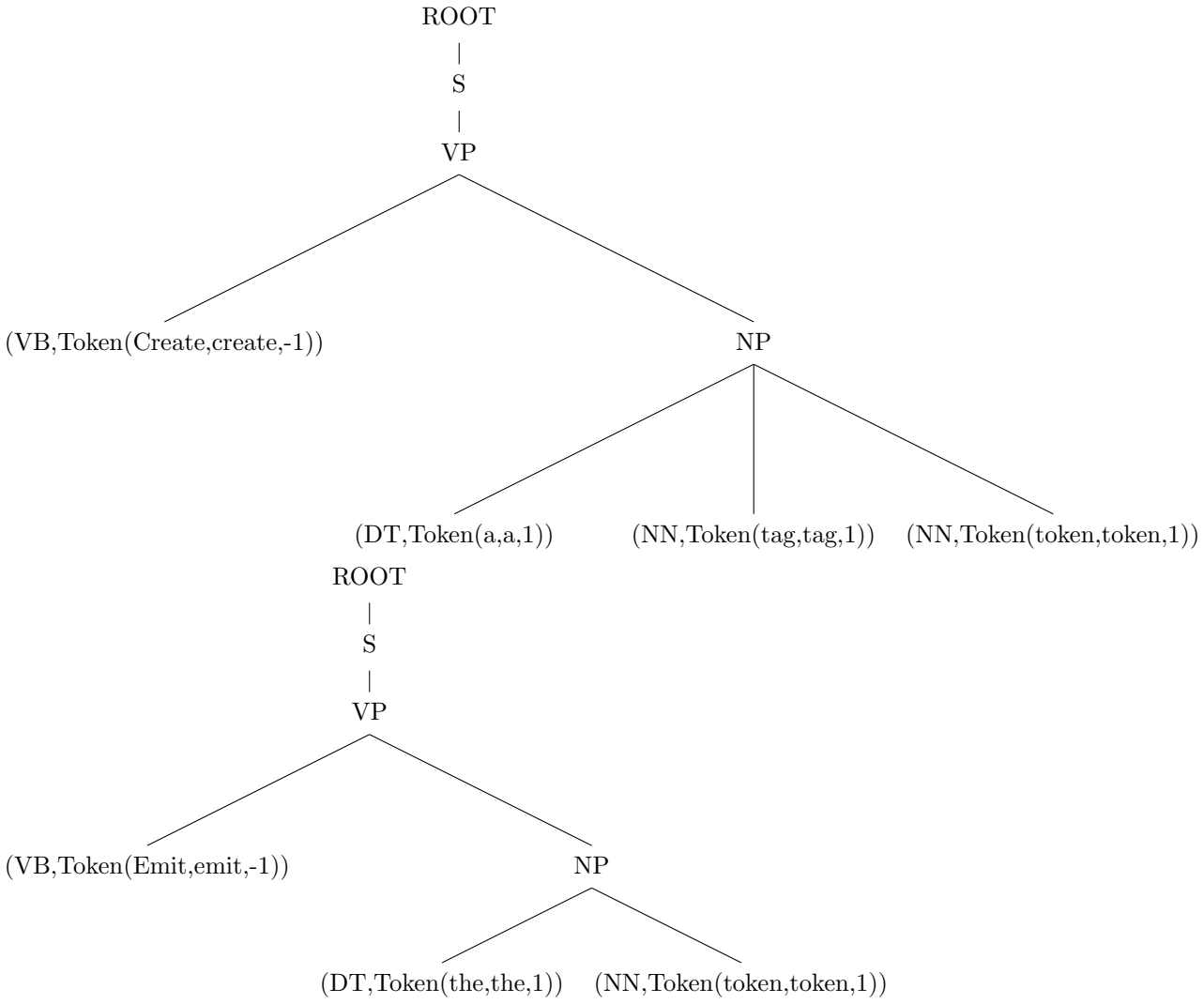
1 つ目は, 自然言語の前処理として適切な解釈がなされるように加えた"you"を取り除くためである. 2 つ目の処理は, カッコの中身に書いてある文章は補足説明が多く, 命令の抽出に必要ないと判断したためである. 3 つ目は, 既に自然言語処理の段階で文章の分割がなされており不要であるから,Tag 構造を簡潔なものにするため取り除く. 4 つ目は, 命令の抽出において, 単語が動詞かどうかを判断できれば十分であるので"VB"に統一することにした.

参照関係の処理

参照関係の出力として,CorefEntity：1 ⇒ [a tag token, the token] が出力される.
構文木を Tag 型に変換する際に, 参照関係を持っている単語の Token の参照番号をその番号とする. 参照関係を持たない単語に関しては参照番号を-1 とする.

変換後の Tag 構造

構文木の処理と参照関係の処理を行った結果.

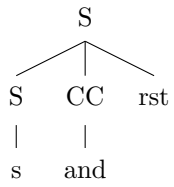


第 6 章

命令の抽出

6.1　Tag 型から Command 型への変換

6.1.1　S ノードの変換

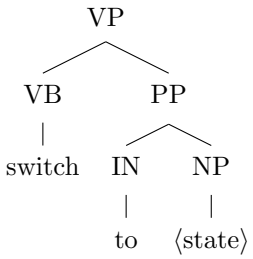


あ

6.1.2　VP ノードの変換

Switch 文

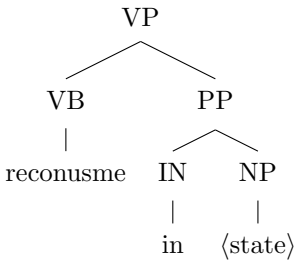
Switch to the ··· state



→Switch(⟨state⟩)

Reconsume 文

Reconsume in the ··· state



→Reconsume(⟨state⟩)

6.1.3 NP ノードの変換

6.2 If 文の処理

6.3 NP ノードから CommandValue 型への変換

NP ノードを CommandValue 型に変換する際, 単純に文字列に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で実装した.

第 7 章

実装

インタープリタを作成した.

7.1 Env

7.2 Command 型

Command 型のそれぞれの値の解釈

Switch(state: CommandValue)

$\langle$ Switch(state), $env\rangle \rightarrow env[nextState \leftarrow \mathcal{C}[\![state]\!]]$

Recomsume(state: CommandValue)

If(bool: Bool, t: CommandList, f: CommandList)

$$\frac{\langle \text{ clist1}, env\rangle \rightarrow env'}{\langle \text{if } b \text{ then clist1 else clist2}, env\rangle \rightarrow env'} \text{ if } \mathcal{B}[\![b]\!] = true$$

$$\frac{\langle \text{ clist2}, env\rangle \rightarrow env'}{\langle \text{if } b \text{ then clist1 else clist2}, env\rangle \rightarrow env'} \text{ if } \mathcal{B}[\![\mathcal{C}[\![b]\!]]\!] = false$$

7.3 Bool 型

And(a: Bool, b: Bool)

CharacterReferenceConsumedAsAttributeVal()

CurrentEndTagIsAppropriate()

IsEqual(a: CommandValue, b: CommandValue)

7.4 Token 型

tagToken(isStart: Boolean, name: String, attributes: List[Attribute]) DOCTYPEToken(systemIdentifier: String, publicIdentifier: String) characterToken()

7.5 Value

CharVal(c: Char) StringVal(string: String) EOFVal StateVal(statename: String) TokenVal(token: Token)

7.6 CommandValue 型

CommandValue 型から Value 型の値を返す関数
C : CommandValue $\rightarrow$ Value

LowerCaseVersion(cVal: CommandValue)

c.toLowerCase if $C[\![\text{cVal}]\!]$ =c: Char or String

NumericVersion(cVal: CommandValue)

Integer.parseInt(c.toString, 16)

NextInputCharacter

$$\begin{cases} \text{CharVal(c)} & \text{inputText.headOption} = \text{Some(c)} \\ \text{EOFVal} & \text{inputText.headOption} = \text{None} \end{cases}$$

CurrentInputCharacter

currentInputCharacter

EndOfFileToken

TokenVal(endOfFileToken())

第 8 章

評価

8.1 HTML5 テスト

字句解析のインタプリターの正しさを検証するために,html5lib-tests [1] の tokenizer のテストデータを用い, テストを行った.

テスト結果

テストファイル名	結果	内容
contentModelFlags.test	24/24	あ
domjs.test	42/58	あ
entities.test	80/80	あ
escapeFlag.test	9/9	あ
namedEntities.test	4210/4210	あ
numericEntities.test	336/336	あ
pendingSpecChanges.test	1/1	あ
test1.test	63/68	あ
test2.test	35/45	あ
test3.test	1374/1786	あ
test4.test	81/85	あ
unicodeChars.test	323/323	あ
unicodeCharsProblem.test	5/5	あ

8.2 問題点

test2.test,test3.test,test4.test が上手くいかなかった原因

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
test2.test	45/45
test3.test	1786/1786
test4.test	85/85

第 9 章

結論

命令の種類が限られており, それぞれ決まった書き方をしていることが多かったので, 構文木の情報のみでも命令の抽出がやりやすかった.(機械的な文字のマッチングでも出来そうではあった.)
しかし今回はやらなかったが, 特に命令の記法が一貫していない場合は係り受け解析を用いたほうが様々な形式の文章に対応できるので良いと思われる.

謝辞

謝辞. 謝辞. 謝辞. 謝辞.

参考文献

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

[2] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[3] Jeanette Pettibone. Penn treebank ii tags, 2020. https://web.archive.org/web/20130517134339/http://bulba.sdsu.edu/jeanette/thesis/PennTags.html.

[4] WHATWG. Html standard, 2020. https://html.spec.whatwg.org/multipage/parsing.html.