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.

過去に行われてきた HTML5 の構文解析の検証に関する研究には, XSS(クロスサイトスクリプティング) 保護機構である XSSAuditor のトランスデューサでのモデル化による有効性の検証 [9] [8] や, HTML5 構文解析仕様の到達可能性の解析 [4] などがある. それらの研究では実装の評価において, 自然言語によって書かれている HTML5 字句解析仕様から, 命令を手作業で抽出している. よって構文解析の検証における実装の負担を減らすために, 仕様書からの命令の抽出, 形式化を自動化したいと考えた. さらに, 仕様の形式化の自動化が出来るようになることによって, 仕様の変更が行われる際, 仕様の定式化を自動化していれば, 変更への対応が楽になるというメリットもある.

仕様書から自然言語処理を用いて命令抽出を試みた研究としては, 機械語命令 ARM の仕様書を対象としたものがあり, そこでは, 自然言語処理の構文木解析などの結果を用いて, 命令の抽出を行っていた. [6]

実装では, 自然言語処理のライブラリとして, スタンフォード大学によって提供されている Stanford CoreNLP [3] を使用した.

（何か書く）

図 1.1 が HTML5 の字句解析仕様の意味解析の概要である.

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.

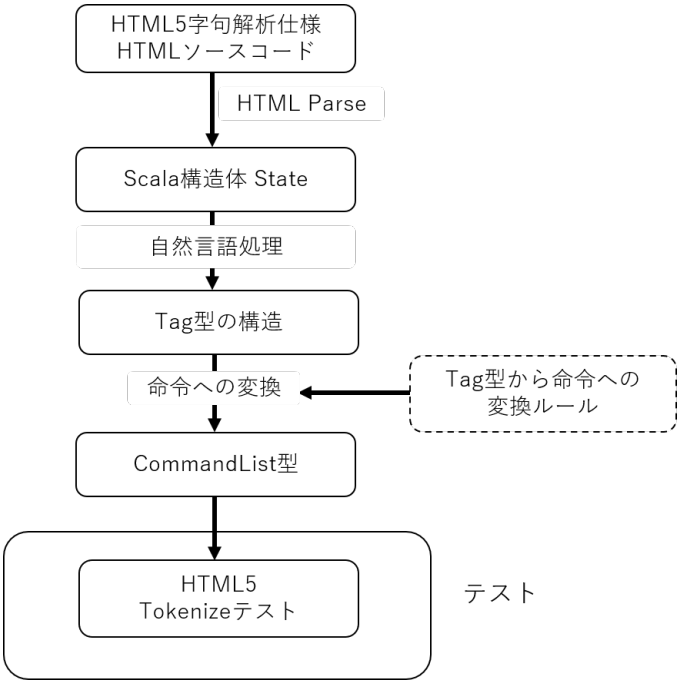

HTML5字句解析仕様
HTMLソースコード
HTML Parse
Scala構造体 State
自然言語処理
Tag型の構造
命令への変換
Tag型から命令への
変換ルール
CommandList型
HTML5
Tokenizeテスト
テスト

図 1.1　流れ（仮）

第 2 章

準備

2.1　節, 句, 品詞タグ

本論文で使用する, いくつかの主要な節, 句, 単語のタグ名とその意味について書く. [5]

節, 句の種類

S　: 文
VP　: 動詞句
NP　: 名詞句
PP　: 前置詞句

単語の種類 (品詞)

VB　: 動詞
NN　: 名詞
IN　: 前置詞, 従属接続詞
DT　: 限定詞
CC　: 等位接続詞

2.2　自然言語処理

自然言語処理とは, 自然言語で書かれている文章に対して, 形態素解析 (品詞タグ付け、単語の原型の取得) や構文解析, 意味解析などをする処理である.

2.2.1　トークン分割, 品詞タグ付け, レンマ化

トークン化では, 文章を単語に分割する. 品詞タグ付けでは, その単語の品詞を調べる. レンマ化とは, 単語の原型を調べるものである.

例 1

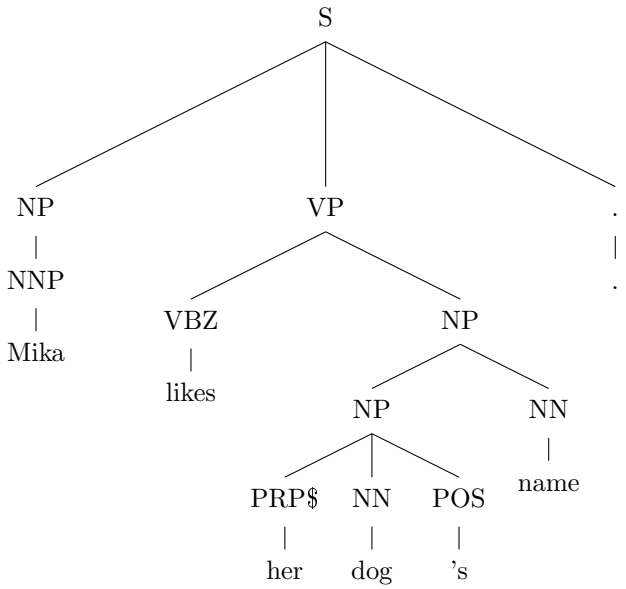
```
Mika likes her dog's name.
```

これをトークン分割, 品詞タグ付け及びレンマ化をすると,
単語/品詞タグ：Mika/NNP likes/VBZ her/PRP$ dog/NN 's/POS name/NN ./.
単語/原型：Mika/Mika likes/like her/she dog/dog 's/'s name/name ./.
となる.

2.2.2 構文木解析

構文木解析とは, 単語を節という単位でグループ化し, その構成を木構造で表現するものである. 構文木解析をすることによって, 文章の構造や単語同士のまとまりを調べることが出来る.

例 1 の “Mika likes her dog's name.” を構文木解析すると.



の様になる.

2.2.3 係り受け解析

係り受け解析とは、節や単語間の関係を調べるものである.
目的語, 修飾語などといったの単語や節の関係性を表す係り受けタグを使用し, それらを関連付ける. 例えば, 単語 B が単語 A の目的語である場合, A $\xrightarrow{\text{目的語}}$ B という風に表記する.

例の, “Mika likes her dog's name.” を係り受け解析すると, 図 2.1 の様になり,
“Mika” が好きなのは, “name”, “name” は “dog” の名前, “dog” は 彼女 (her) の犬である
という様に, 単語の目的語, 所有しているものを示してる単語が解析されている.

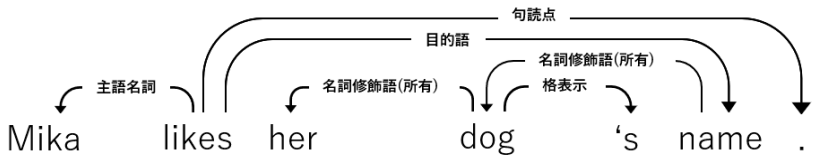


図 2.1　係り受け解析

2.2.4　固有表現抽出

固有表現抽出とは, 文章内に数字や時間、アドレス、人間、地名を意味する単語があった場合,「この単語は地名を表すものである」という風に, 固有な表現を解析するものである.

例 1 の “Mika likes her dog’s name.” の場合は, “Mika” が “人間” を表す単語であるということが固有表現抽出によって出力される.

2.2.5　参照関係の解析

参照関係の解析とは, 文章内で同じものを指し示している単語が複数ある時, それらを関連付けさせ, 抽出するものである.
これによって, “it” や “he” などの指示語の指し示すものを見つけることなどが出来る.

例 1 の “Mika likes her dog’s name.” の場合は, “Mika”, “her” が同一のものであることが参照関係の解析によって分かる.

第 3 章

HTML5 字句解析仕様

まず, 自然言語処理の適用対象である, HTML5 の字句解析仕様について概要を述べていく.

3.1　概要

HTML5 字句解析仕様は WHATWG community の web サイトから得られる. [7]
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おり, それぞれの状態は以下の図 3.1 の形式で書かれている.

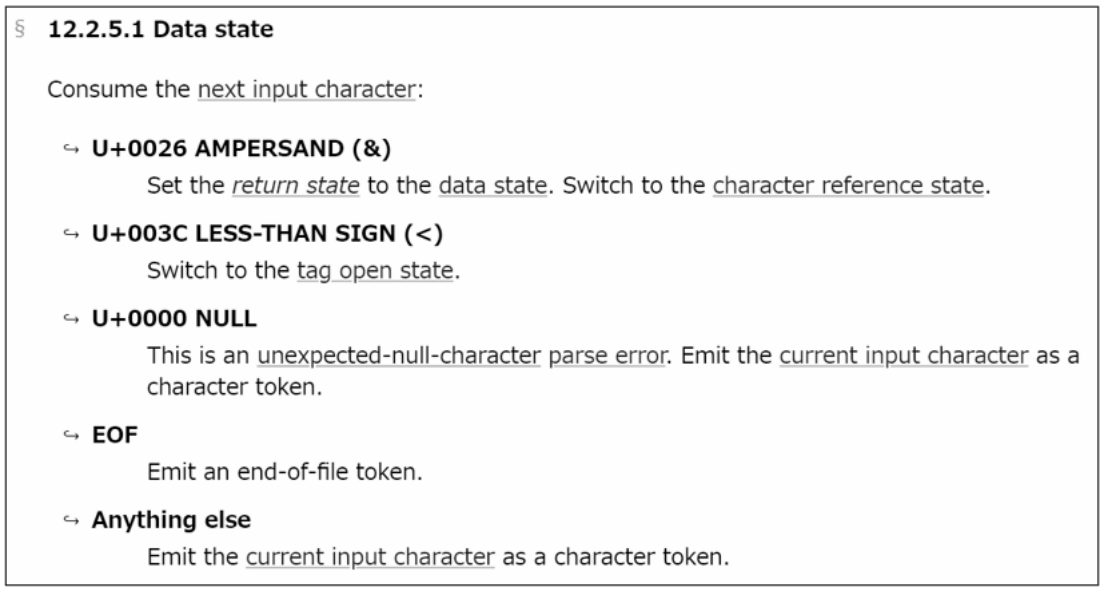


図 3.1　HTML5 字句解析仕様書（仮）なくていいかも

字句解析トークン

HTML5 字句解析器は, HTML5 で書かれている文章をその文法の構成単位であるトークンに分解するプログラムである.

HTML5 字句解析器により出力されるトークンは 5 つの種類がある．文書で利用する HTML や XHTML のバージョンを表すトークンである, DOCTYPE トークン (DOCTYPE token), “<!-- -->” などでコメントアウトした文章を値として持つコメントトークン (comment token), ‘<’ と ‘>’ で囲まれている, HTML のタグを表す開始タグトークン (start tag token) と終了タグトークン (end tag token), 文字の情報を表す文字トークン (character token), 文章の終了を表す EOF トークン (end-of-file token) が存在する．

EOF トークン以外はそれぞれ値を持っている．
DOCTYPE トークンは DOCTYPE の名前, 公開識別子 (public identifier) とシステム識別子 (system identifier), 強制互換モードのフラグ (force-quirks flag) の要素を持っている．
開始, 終了タグトークンはタグの名前, 属性の集合 (attributes), セルフクロージングタグかどうかのフラグ (self-closing flag) の要素を持っている．
文字トークンは文字のデータを持つ．コメントトークンはコメント内容の文字列データを持つ．

具体的なトークンへの分解の例として, HTML5 文章

```
<!DOCTYPE html> <!-- ABCDEFG --> <p> hello </p>
```

をトークンに分解すると,
DOCTYPE トークン (名前: html, 公開識別子: null, システム識別子: null, 強制互換モードのフラグ: off)
コメントトークン (ABCDEF)
開始タグトークン (名前: p, 属性: 無し, セルフクロージングタグのフラグ: off)
文字トークン (hello)
終了タグトークン (名前: p, 属性: 無し, セルフクロージングタグのフラグ: off)
となる．

字句解析器により出力されたトークンは DOM ツリーを構成する次の構文解析のステップに使われる．

字句解析器の環境の変数

HTML5 字句解析器は return state や一時バッファなどの様々な変数を持ち, 様々な命令により, 環境の変数の値を変化させていき, 動作する．

3.2 字句解析器の動作例

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開始タグトークンを作る．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, 開始タグトークンの名前に’a’ を付け足す．
4. 文字’>’ を消費し, 開始タグトークンを排出し, Data state に遷移する．
5. 文字’b’ を消費し, 文字トークン (’b’) を排出する．

6. 文字'c' を消費し, 文字トークン ('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終了タグトークンを作る. 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, 終了タグトークンの名前に'a' を付け足す.
11. 文字'>' を消費し, 終了タグトークンを排出し, Data state に遷移する.
12. EOF トークンを排出する.

動作の結果として,
開始タグトークン (名前: "a", 属性: [], セルフクローズフラグ: off), 文字トークン ('b'), 文字トークン ('c'),
終了タグトークン (名前: "a"), EOF トークン
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, 文字トークン ('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開始タグトークンを作る. 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, 開始タグトークンの名前に'a' を付け足す.
5. 文字'b' を消費し, 開始タグトークンの名前に'b' を付け足す.
6. "eof-in-tag" 構文エラーを出す. EOF トークンを排出する.

動作の結果として,
"eof-in-tag" 構文エラーと,
文字トークン ('a'), EOF トークンが排出される.

第 4 章

抽出の形式（モデル）

HTML5 の仕様書から命令を抽出するため, BNF の記法を用いて, 命令を形式化する.

4.1　1 状態の目標とする形

4.2　抽出する命令の形式

まず命令の形式の, メタ変数, 型を以下で定める.

c : Command	･･･命令文
b : Bool	･･･if文の命令文の条件部分の文
cval: CommandValue	･･･命令文が引数に持つ値
ival : InplementVariable	･･･命令文が引数に持つ代入される変数の種類

cval, ival は仕様書において, 字句解析器の変数や値として出てくるものであり, ival は代入される対象, cval は代入するものを表している.
メタ変数は, c_1, c_2 のように, 添え字つけられていてもよいものとする.

それぞれの型の構造は以下のように定める.

Command 型

c ::=	If(b, $cList_1$, $cList_2$)	// if b then $cList_1$ else $cList_2$
|	Ignore()	// 何もしない処理
|	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(string, cval)	// トークン cval を新たに作り, 変数 string にトークン cval を代入する
|	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

Bool 型

b ::=	And(b_1, b_2)	// b_1 かつ b_2 である
|	Or(b_1, b_2)	// b_1 または b_2 である
|	Not(b)	// b でない
|	CharacterReferenceConsumedAsAttributeVal()	// CharacterReferenceCode が属性の値として消費されているか
|	CurrentEndTagIsAppropriate()	// EndTagToken が適切なものであるか
|	IsEqual($cval_1$, $cval_2$)	// $cval_1$ と $cval_2$ の値が等しい
|	AsciiCaseInsensitiveMatch($cval_1$, $cval_2$)	// $cval_1$ と $cval_2$ が文字列であるとき, 大文字, 小文字の差を無視し $cval_1$ と $cval_2$ の値が等しい

CommandValue 型

```
cval :: =   StateName(string)          // 状態名 string
        |   ReturnState                // 字句解析器の変数：return state
        |   TemporaryBuffer            // 字句解析器の変数：temporary buffer (一時バッファ)
        |   CharacterReferenceCode     // 字句解析器の変数：character reference code
        |   NewStartTagToken           // 初期状態の開始タグトークン
        |   NewEndTagToken             // 初期状態の終了タグトークン
        |   NewDOCTYPEToken            // 初期状態の DOCTYPE トークン
        |   NewCommentToken            // 初期状態のコメントトークン
        |   CurrentTagToken            // 一番新しく作られたタグトークン
        |   CurrentDOCTYPEToken        // 一番新しく作られた DOCTYPE トークン
        |   CurrentAttribute           // 一番新しく作られたタグトークンの属性 (attribute)
        |   CommentToken               // 一番新しく作られたコメントトークン
        |   EndOfFileToken             // EOF トークン
        |   CharacterToken(cval)       // 中身が cval の文字トークン
        |   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
        |   Substitute(string, cval)   // cval (副作用として変数 string に cval を代入する)
        |   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
```

InplementVariable 型

ival :: =	IReturnState	// 字句解析器の変数：return state
	ITemporaryBuffer	// 字句解析器の変数：temporary buffer
	ICharacterReferenceCode	// 字句解析器の変数：character reference code
	ICurrentTagToken	// 一番新しく作られたタグトークン
	ICurrentDOCTYPEToken	// 一番新しく作られた DOCTYPE トークン
	ICurrentAttribute	// 一番新しく作られたタグトークンの属性 (attribute)
	ICommentToken	// 一番新しく作られたコメントトークン
	IVariable(string)	// 変数 string
	INameOf(ival)	// タグトークン, 属性である ival の名前
	IValueOf(ival)	// 属性 ival の値
	IFlagOf(ival)	// DOCTYPE, タグトークン ival のフラグ
	SystemIdentifierOf(ival)	// DOCTYPE トークン ival のシステム識別子
	PublicIdentifierOf(ival)	// DOCTYPE トークン ival の公開識別子

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

4.3 命令形式の例

自然言語の文から, 上記の形式への変換の例をいくつか記す.

状態 Data_state へ遷移するという意味の文に関して,
Switch to the Data_state.
⇒ Switch(StateName(Data_state))
状態名を意味する StateName(Data_state) という CommandValue 型の値を持つ, Switch という遷移を意味する Command 型の値に変換する.

また, 現在のタグトークン名に, 現在消費した文字を小文字にしたものを付け足すという意味の文に関して,
Append the lowercase version of the current input character to the current tag token's tag name.
⇒ Append(LowerCase(CurrentInputCharacter), INameOf(CurrentTagToken))
第 1 引数に "現在消費した文字の小文字" を意味する CommandValue 型の値, 第 2 引数に "現在のタグトークン名" を意味する ImplementVariable 型の値を持つ Command 型の Append という値に変換される.

この自然言語の文章から上記の形式への変換の過程を 5, 6 章で説明していく.

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オートマトンとしての状態が 80 存在する．そして, 80 個の状態のうち, 77 の状態は同じような構造で命令が記述してある．しかし, 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たり, 表を参照する必要が出てきたりするので, 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思われた.

よって本論文では自然言語処理の適用の対象を 80 の状態うち, 上記の 3 状態を除く 77 の状態に絞ることにした. 尚, テストする際は残りの 3 状態は手動で実装することにした.

また, HTML5 仕様書内の Note や Example 等の補足説明は無視する.

5.2　対象の前処理

HTML5 字句解析仕様書は HTML によって構造的に書かれているので, 仕様書の文章をそのままの状態で丸ごと自然言語処理させると上手くいかない. よって, 自然言語処理の適用の前段階の処理として, 以下を行うことにした.

字句解析仕様書の HTML のソースコードを仕様書解析の入力とし, 前処理の第 1 段階として, そのソースコードを HTML パーサーに通し, 構造を認識し, Scala で定義した構造体に置き換え, 自然言語処理を適用したい部分の文章を抜き出す. そして次の段階として, その文章に対して自然言語処理が適切に行われるように, 特定の文字列の置き換えを行う.

5.2.1　構造体の処理

仕様書の HTML ソースコードをみると, 1 状態あたり, 図 5.1 のように書かれている.

Listing 5.1　HTML ソースコード

```
<h5> <dfn> 状態名 <\dfn> <\h5>
<p> 文字マッチングする前の処理 <\p>
<dl>
    <dt> 文字1 <\dt>
    <dd> 文字にマッチした時の処理1 <\dd>
    ...
```

```
    <dt> 文字n <\dt>
    <dd> 文字にマッチした時の処理n <\dd>
<\dl>
```

これを HTML パーサーのライブラリ jsoup [2] を使って,「状態名」,「文字マッチングする前の処理」,「文字 i とその文字にマッチした時の処理のリスト」をもつ, 図 5.2 の構造体 State に置き換える.

Listing 5.2 State の構造体

```
case class State(name: String, prevProcess: String, trans: List[Trans])
case class Trans(character: String, process: String)
```

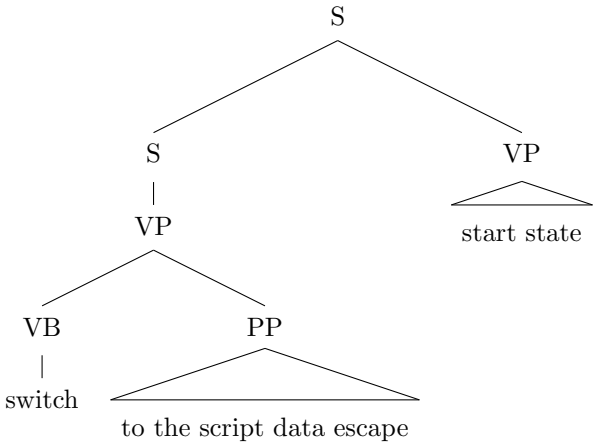
具体的な処理としては, “h5”, “dfn” タグで囲まれている文を “状態名” として取り出し, “h5” タグ直後の “p” タグ内の文を “文字マッチングする前の処理” として取り出す. 文字のマッチングの処理は, “dl” タグ内に書かれており, その中の “dt” タグから “マッチングの文字 i”, その直後にある “dd” タグから “文字 i にマッチした時の処理” を取り出す.
自然言語処理は,「文字マッチングする前の処理」と「文字 i とその文字にマッチした時の処理のリスト」の処理の部分に対して行う.

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, 様々な原因によりトークンの分割や品詞解析が適切な形で解釈されないので, 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

“script data escape start state” に遷移するという命令文 “Switch to the script data escape start state.” は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.
また, 状態名の中にカッコや “-” がある場合に関しても同じようなことが起こる.

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, 仕様書内に出てくる状態名に関して以下の処理をし, 置き換えを行う.

- 空白 及び “-” を “_” に置き換える.
- “(” , “)” を除く.
- 先頭を大文字にする.

先頭を大文字に置き換えるのは, 仕様書内の状態名の先頭が大文字と小文字の場合が混ざっており, 統一させるため.
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. これに対して自然言語処理を行うと, トークン分割において “U”, “+xxxx” と 2 つのトークンに分割されてしまう. 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

自然言語処理の出力を確認すると, 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.
“Reconsume” の他に, 動詞が “Emit”, “Flush”, “Append”, “Multiply” も同じようなことが起こる. また, Stanford CoreNLP は命令文の解釈が普通の文より上手くいかないことがある. よって命令文の解釈が上手くいかない文の動詞 (“Switch”, “Append”, “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.

- 文字列 “Switch”, “Reconsume”, “Emit”, “Flush”, “Append”, “Add”, “Multiply” の前に “you” を加える.

例: “Reconsume in the data state.” ⇒ “you Reconsume in 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　命令の抽出元 (Tag 型) への変換

命令の抽出をするため, その元となる, 多分木の木構造のデータ型である Tag 型をプログラミング言語 Scala で定義し, Stanford CoreNLP を用いての自然言語処理から得られる情報のうち, 単語の原型の解析, 構文木解析, 参照関係の解析の情報を Tag 型への変換に使用した.

5.3.1　Tag 型

図 5.3 に Tag 型の定義を載せる.
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 型は順に 単語, 単語の原型, 参照関係の番号の情報を持つ.

Listing 5.3　Tag の定義

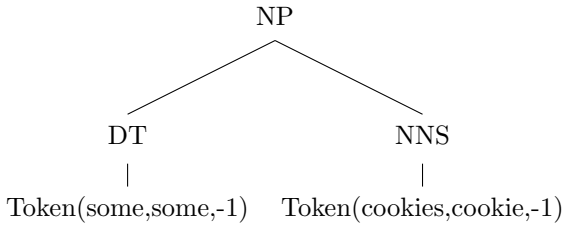
```
trait Tag
case class Node(node: NodeType, list: List[Tag]) extends Tag
case class Leaf(leaf: LeafType, token: Token) extends Tag
case class Token(word: String, lemma: String, coref: Int) extends Tag
trait NodeType
case object S extends NodeType
case object NP extends NodeType
case object VP extends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

例

Tag 型の値

```
Node(NP, List(Leaf(DT, Token(some,some,-1)), Leaf(NNS, Token(cookies,cookie,-1))))
```

を木構造で表記すると,



となる.

5.3.2 Tag 型への変換

単語の原型の解析, 構文木解析, 参照関係の解析の情報を Tag 型への変換の方法を書く.

構文木の処理

基本的には自然言語処理の構文木解析で出力された木の形をそのままの状態で同じ木構造である 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.

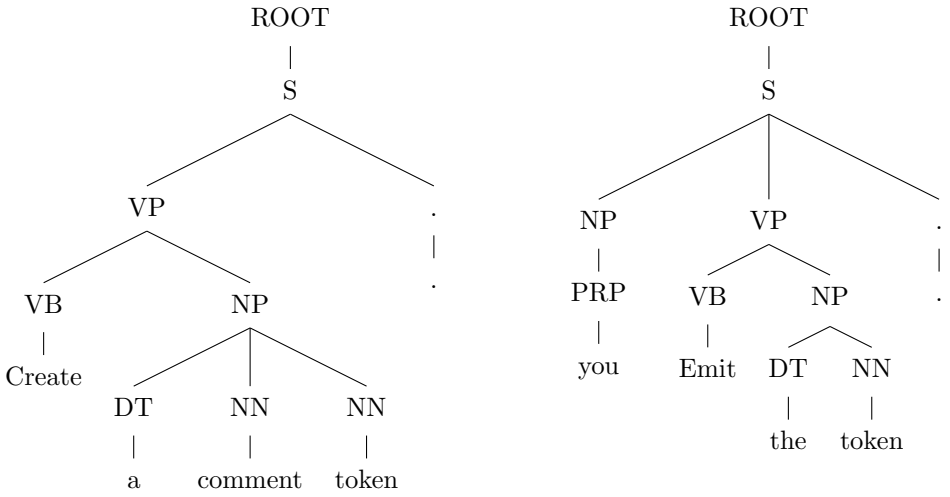
Leaf 型が持つ Token に関しては, 単語の原型の部分にレンマ化の処理で得た原型の情報を代入し, 参照番号の部分で下記の参照関係の処理を行い, 値を代入する.

参照関係の処理

参照関係の出力として, CorefEntity：1 ⇒ [a new tag token, the token] という風に, 参照関係がある単語と, 参照の番号 (この場合では 1) が得られる.
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 型が持つ参照番号をその番号とし, 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.

5.4 Tag への変換例

文章 “Create a comment token. Emit the token.” から Tag 型に変換する例を示す.
まず, 文字の置き換えの前処理を行うと, “Create a token. you Emit the token.” となる.
次に, これに対して自然言語処理を行うと,
単語の原型は,
Create/create a/a comment/comment token/token ./.
you/you Emit/emit the/the token/token ./.
構文木解析は,



参照関係の解析では, “a comment token” と “the token” が同じものであると出力される.

そして, 構文木の処理と参照関係の処理を行い, これを Tag 型への変換をすると,
図 5.1 のようになる.

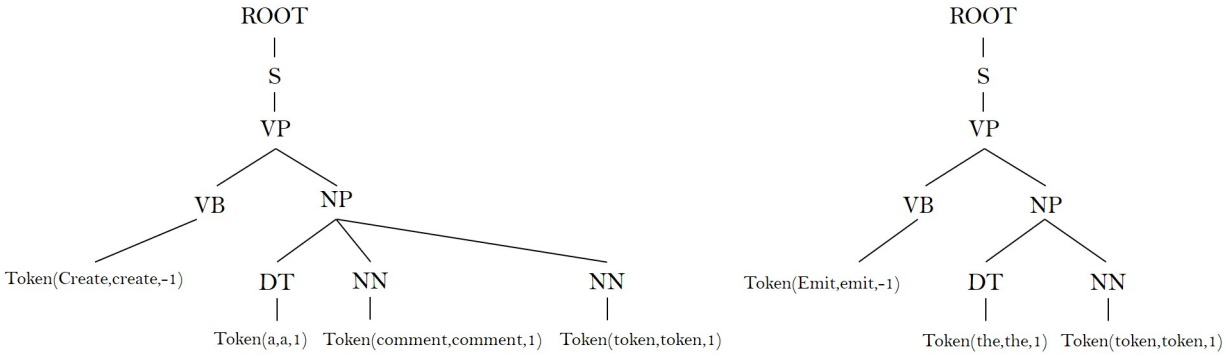


図 5.1　Tag 型

第 6 章

命令の抽出

5 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, 4 章で定式化した形での命令の抽出を行う.

6.1 Tag 型から Command への変換

Tag 型の形に関してのパターンマッチングをして, CommandList 型の値へ変換する. Tag 型の Node の NodeType が S, VP, NP(文, 動詞句, 名詞句) である場合に分け, 変換を行った. 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.

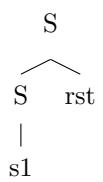
この変換を関数として表記する.
NodeType が S である Node から, CommandList 型へ変換する関数を $\mathcal{T}_S$, NodeType が VP である Node から, CommandList 型へ変換する関数を $\mathcal{T}_{VP}$, NodeType が NP である Node から, CommandList 型へ変換する関数を $\mathcal{T}_{NP}$ と書く.

6.1.1 文 (S ノード) の変換 $\mathcal{T}_S$

$\mathcal{T}_S$ は以下のパターンマッチを行い, マッチしたものに応じた CommandList 型の値を返す.

子ノードの先頭が S ノード

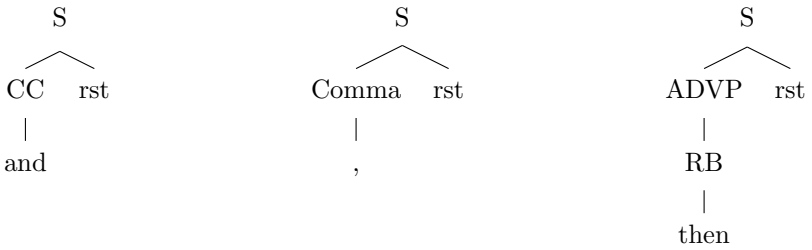
S の子ノードのリストの先頭が S ノードの場合, つまり Tag 型が



という形 (rst は Tag 型のリスト)
$\Rightarrow \mathcal{T}_S$(Node(S, s1)) ++ $\mathcal{T}_S$(Node(S, rst)) を返す.

子ノードの先頭が and, then, カンマ

S の子ノードのリストの先頭が葉 “and”, “then”, カンマの場合, つまり Tag 型が



という形 (rst は Tag 型のリスト)

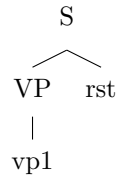
⇒ $\mathcal{T}_S$(Node(S, rst)) を返す. (“and”, “then”, カンマは無視し, 子ノードの残りのノードに関して $\mathcal{T}_S$ を適用する.)

子ノードが空リスト

S の子ノードの空リストの場合, Nil を返す.

子ノードの先頭が VP ノード

S の子ノードのリストの先頭が VP ノードの場合, つまり Tag 型が



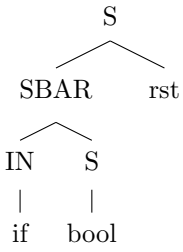
という形 (rst は Tag 型のリスト)

⇒ $\mathcal{T}_{VP}$(Node(VP,vp1)) ++ $\mathcal{T}_{\mathcal{S}}$(Node(S,rst)) を返す.

(Node(VP, vp) に関して, 動詞句の変換を行う.)

If 文のマッチ

S の子ノードのリストの先頭が条件文, つまり Tag 型が,



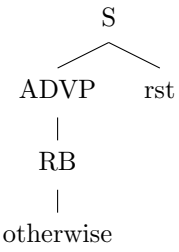
の形 (rst は Tag 型のリスト)

⇒ IF_($\mathcal{T}_{\mathcal{B}}$(bool)) :: $\mathcal{T}_{\mathcal{S}}$(Node(S,rst))

を返す.

子ノードの先頭が Otherwise

S の子ノードのリストの先頭が “Otherwise” のとき, つまり Tag 型が,

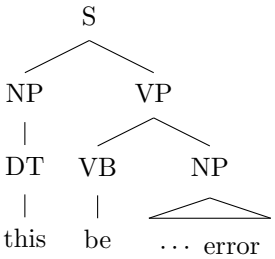


の形 (rst は Tag 型のリスト)

⇒ Otherwise_() :: $\mathcal{T}_S$(Node(S,rst)) を返す.

Error 文のマッチ

Tag 型が,



の形

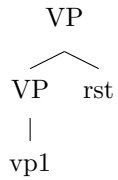
⇒ Error(··· error) :: Nil を返す.

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

NodeType が VP である Node(動詞句) から, Command 型のリストへ変換する関数を $\mathcal{T}_{VP}$ と置く. $\mathcal{T}_{VP}$ は以下のパターンマッチを行い, マッチしたものに応じた Command 型の List を返す.

子ノードの先頭が VP ノード

VP の子ノードのリストの先頭が VP ノードの場合, つまり Tag 型が



という木構造の形の時, $\mathcal{T}_{VP}$(Node(VP,vp1)) ++ $\mathcal{T}_{VP}$(Node(VP,rst)) を返す.

子ノードの先頭が and, カンマ (VP を繋ぐ単語)

VP の子ノードのリストの先頭が葉 “and”, カンマの場合, つまり Tag 型が

という木構造の形の時, $\mathcal{T}_{VP}$(Node(VP, rst)) を返す.

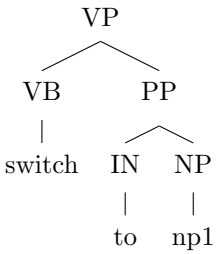
子ノードが空リスト

VP の子ノードの空リストの場合, Nil を返す.

Switch 文のマッチ

(元の文：Switch to the $\cdots$ state)

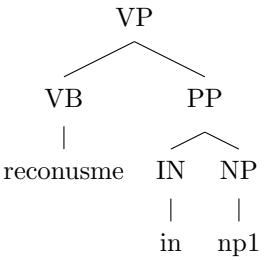
Tag の形が,



の時, Switch($\mathcal{T}_{NP}$(np1)) :: Nil を返す.

Reconsume 文のマッチ

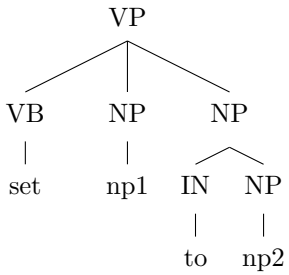
Reconsume in the $\cdots$ state



の時, Reconsume($\mathcal{T}_{NP}$(np1)) :: Nil を返す.

Set 文のマッチ

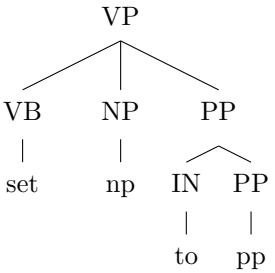
変数に値を代入する.



の場合

⇒ Set($\mathcal{T}_{NP}$(Node(NP, np1)), $\mathcal{T}_{NP}$(Node(NP, np2))) を返す.

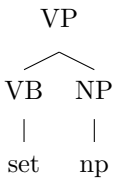
変数の状態を変える.



の場合

⇒ pp が “on” だったら, Set($\mathcal{T}_{NP}$(Node(NP, np)), On) を返す.

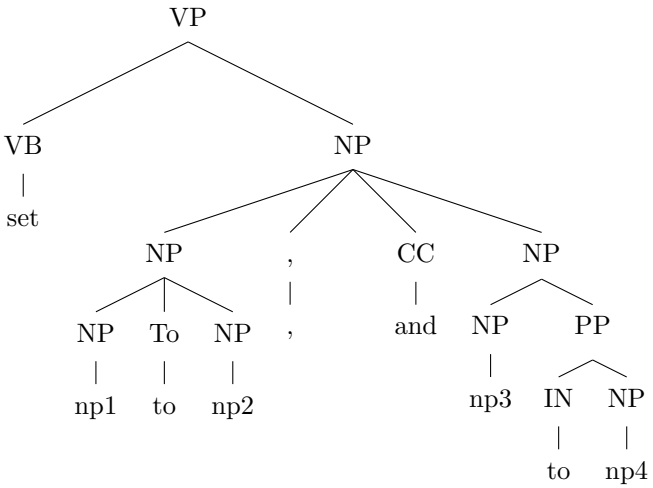
■個別に対応した Set 文のマッチ “Set the self-closing flag of the current tag token.” のような状態の切り替え先を示す “to on” が省略されている場合もあった. このような文の構文木は



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) とさせた.

“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, 個別に対応した.

木構造が,



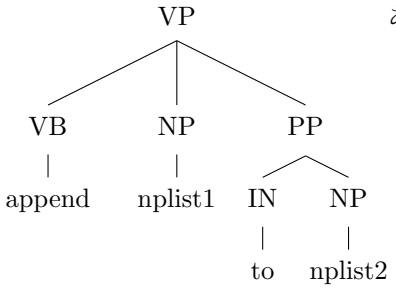
のとき,

[Set($\mathcal{T}_{NP}$(Node(NP, np1)), $\mathcal{T}_{NP}$(Node(NP, np2))), Set($\mathcal{T}_{NP}$(Node(NP, np3)), $\mathcal{T}_{NP}$(Node(NP, np4)))]

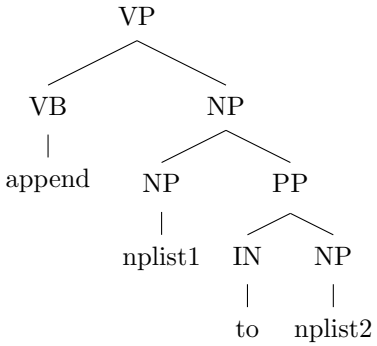
を返す.

AppendTo 文のマッチ

木構造が,



あるいは

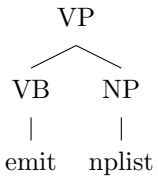


の場合

⇒ AppendTo($\mathcal{T}_{NP}$(Node(NP, nplist1)), $\mathcal{T}_{NP}$(Node(NP, nplist2))) を返す.

Emit 文のマッチ

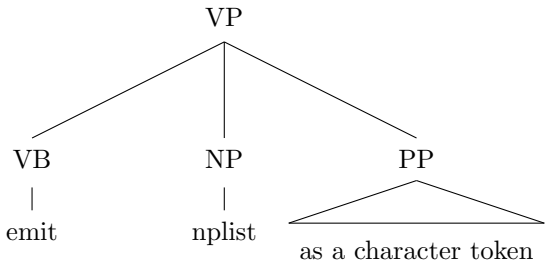
木構造が,



の場合

⇒ Emit($\mathcal{T}_{NP}$(Node(NP, nplist))) を返す.

木構造が,

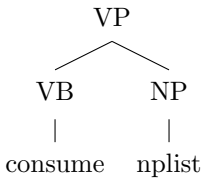


の場合

⇒ Emit(CharacterToken($\mathcal{T}_{NP}$(Node(NP, nplist)))) を返す.

Consume 文のマッチ

木構造が,

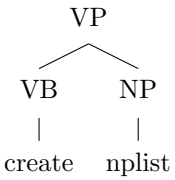


の場合

⇒ Consume($\mathcal{T}_{NP}$(Node(NP, nplist))) を返す.

Create 文のマッチ

木構造が,



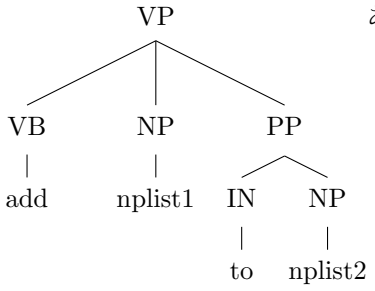
の場合

nplist の単語に番号 n の参照関係が存在 ⇒ Create("x_n", $\mathcal{T}_{NP}$(Node(NP, nplist))) を返す.

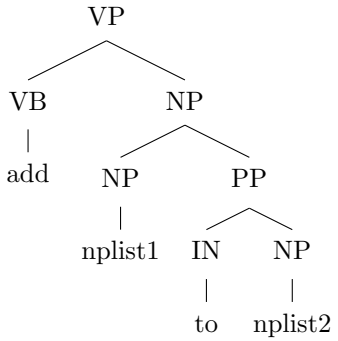
そうでない ⇒ Create("", $\mathcal{T}_{NP}$(Node(NP, nplist))) を返す.

AddTo 文のマッチ

木構造が,



あるいは

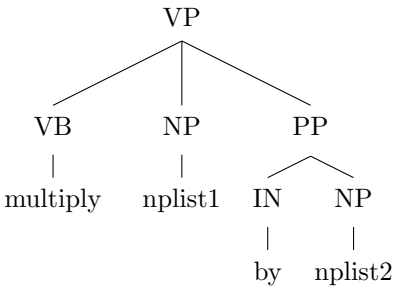


の場合

⇒ AddTo($\mathcal{T}_{NP}$(Node(NP, nplist1)), $\mathcal{T}_{NP}$(Node(NP, nplist2))) を返す.

MultiplyBy 文のマッチ

木構造が,

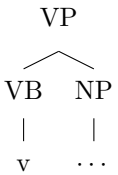


の場合

⇒ MultiplyBy($\mathcal{T}_{NP}$(Node(NP, nplist1)), $\mathcal{T}_{NP}$(Node(NP, nplist2))) を返す.

Ignore, FlushCodePoint 文のマッチ

木構造が,



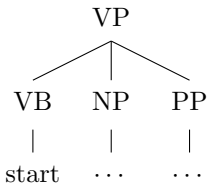
の場合

v が "ingore"の時, Ignore() を返す.

v が "flush"の時, FlushCodePoint() を返す.

StartAttribute 文のマッチ

木構造が,

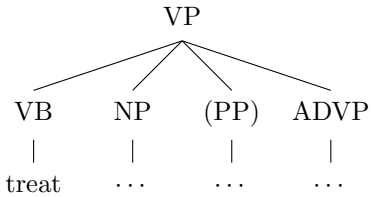


の場合

⇒ StartAttribute() を返す.

TreatAsAnythingElse 文のマッチ

木構造が,



の場合

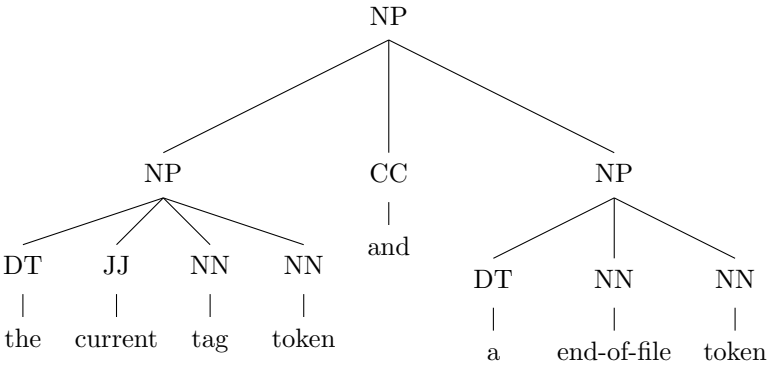
⇒ TreatAsAnythingElse() を返す.

6.1.3　名詞句 (NP ノード) の変換 $\mathcal{T}_{NP}$

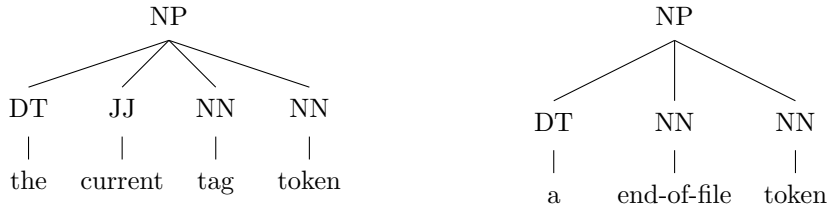
名詞の列挙の分解

“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．

CommandValue 型, ImplementVariable 型への変換

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”))

6.2　If 文の処理

仕様書内の条件分岐の文に関して, Command 型のリストに変換する際, 一部手を加えた．

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
$\cdots$ command_m
Otherwise_()
command_{m+1}
$\cdots$ 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,

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.

6.3 命令への変換の例

例 5.4 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 6.1 のように置く. (点線部分)

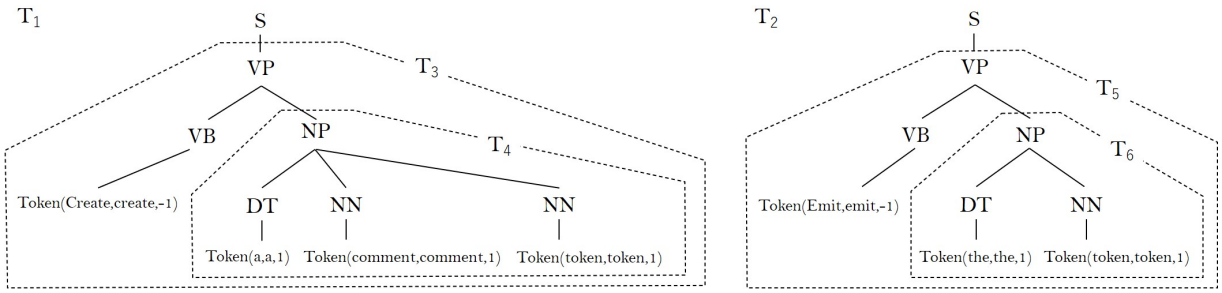


図 6.1 例 5.4 で変換された Tag 型

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

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

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

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

6.4 例 2

§ **13.2.5.9 RCDATA less-than sign state**

Consume the next input character:

↪ **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.

図 6.2 RCDATA less-than sign state

これに 5, 6 章の処理をし, pState の形にすると, pState(stateName: String, prevProcess: List[Command], trans: List[(String, List[Command])])
stateName = RCDATA_less_than_sign_state
prevProcess =
[Consume(NextInputCharacter)]
trans =
[
(U+002F SOLIDUS (/),
[Set(ITemporaryBuffer, CString(“”)), Switch(RCDATA_end_tag_open_state)]),
(Anything else,
[Emit(CString(“<”)), Recomsume(RCDATA_state)])

]

となる.

第 7 章

実装

6 章で HTML5 字句解析仕様から抽出し, 形式化したもの使って HTML5 の字句解析のインタープリタを作成した.

7.1 概要

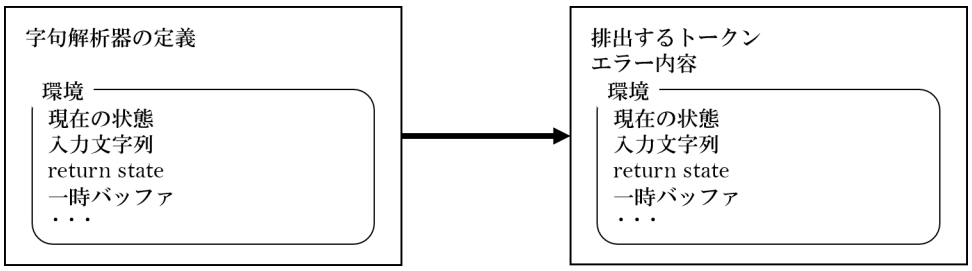


図 7.1 インタープリタ概略

形式化した命令をもとに動かす.

7.2 インタープリタの実装の詳細

7.2.1 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.

Value 型

インタープリタで扱う値の型

Listing 7.1 Value 型

```
IntVal(int: Int)
BooleanVal(boolean: Boolean)
CharVal(c: Char)
StringVal(string: String)
EOFVal
StateVal(statename: String)
```

```
    TokenVal(token: Token)
```

7.2.2 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.

第 8 章

実装の評価

8.1 HTML5 テスト

html5lib-tests [1] の tokenizer のテストデータを用い, インタープリタのテストを行った.

テスト結果

テストファイル名	結果	テスト内容
contentModelFlags.test	24/24	
domjs.test	50/58	
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

8.2 思ったこと

命令抽出に関しては, 手作業でやった部分が多いので上手くいったと思う.

8.3 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
domjs.test	57/58
test2.test	45/45
test3.test	1786/1786
test4.test	85/85

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.

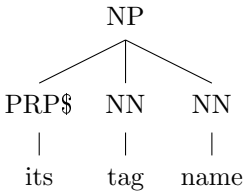
第 9 章

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可能だった. (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.

例えば, Tag 型から命令の型である Command 型へ変換する際, 構文木のマッチングで, “Emit the current input character as a character token.” といった文がある. この文のように “current input character” に “as a character token” のような補足的な情報が加わると, 複数種類のパターンマッチ文を書く必要が出てきており, 命令抽出の対象の記法が比較的一貫したので煩雑さは抑えられたが, 係り受け解析を用いたほうが簡潔にできると感じた. that attribute's は name と value

NP ノードから CommandValue 型への変換する際, 係り受け解析を使用したほうが楽だと思った. “its tag name” とか, 構文木解析だと,



という結果が得られるが,
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

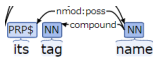


図 9.1　example

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.

[2] Jonathan Hedley. jsoup: Java html parser, 2020. https://jsoup.org/.

[3] 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.

[4] Yasuhiko Minamide and Shunsuke Mori. Reachability analysis of the html5 parser specification and its application to compatibility testing. *FM 2012: Formal Methods*, 2012.

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

[6] Anh V. Vu and Mizuhito Ogawa. Formal semantics extraction from natural language specifications for arm. *Formal Methods–The Next 30 Years*, 2019.

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

[8] 小林 孝広. 交代性オートマトンを用いたトランスデューサの包含関係の保守的検査. 東京工業大学　学士論文, 2019.

[9] 芹田 悠一郎. トランスデューサによる XSS Auditor の有効性の分析. 東京工業大学　学士論文, 2017.