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

HTML5 は, 現代社会にて広く普及している Web ページを作成するためのマークアップ言語のひとつである. HTML5 文書はその描画の処理として, まず字句解析器によって文書が Token という単位に分解され, 構文解析器によってその Token 列から DOM ツリーを作成し, 描画を行う.

HTML5 の構文解析を対象とした研究として, 過去に 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, テストの自動生成 [5] などが行われている. それらの研究では実装段階において, 自然言語によって記述されている HTML5 の構文解析仕様から, 手作業でその命令, 動作を抽出している. そこで, このような研究における手作業による翻訳の負担を減らすために, 仕様書からの命令の抽出, 形式化を自動化することに意義を感じた. よって, 自然言語をコンピュータで処理できる, 自然言語処理の技術を用いて, HTML5 の字句解析仕様から命令を抽出する方法を模索し, それを自動化することを考えた.

自然言語処理を用いて仕様書からの自動形式化を試みた研究としては, 仕様書の構造や自然言語処理による単語の品詞タグ付けの情報を使い, 仕様書に記述されている命令を抽出する研究 [2] や, CPU アーキテクチャの一種である ARM の機械語仕様書を対象とした, その仕様書の意味論抽出行う研究 [6] が行われている. ARM を対象とした研究では, 自然言語処理による構文木解析などの結果を用いて, 自然言語仕様から形式的な意味論を抽出するメソッドを示している.

本研究では, HTML5 の字句解析仕様に対して, その命令の形式化をし, 自然言語処理のライブラリの 1 つである, Stanford CoreNLP [4] を使い, 仕様書に自然言語処理を適用し, 自動形式化する方法を考えた. その方法として, まず, 自然言語処理の構文解析や意味解析の結果を使い, 命令を抽出するための型である `Tag` 型へ変換し, そして, 変換された `Tag` 型から木構造の形のマッチングなどを用いて形式化した命令に変換した. その際, 仕様書の文章に対して直接自然言語処理を適用すると, 構文解析や意味解析の結果が適切なものが得られないことがあったので, 自然言語処理を適用する前の処理として, 特定の文字列の置き換えをするなどしたり, 構文木解析の結果を用いて命令を抽出する際も, 条件分岐の文など、命令の抽出の仕方に関する工夫を行った. そして最後に, 字句解析のインタプリタを作成し, HTML5 の字句解析のテストをして, 仕様書に書いてある通りの意味の命令を抽出出来たことを確認できた.

本論文では, まず 2 章で自然言語処理に関する基礎知識を述べる. 次に 3 章で HTML5 の字句解析器の仕様や具体的な動作について述べる. そして 4 章で抽出する命令の形式化をし, 5 章で HTML5 字句解析仕様への自然言語処理の適用の方法について述べ, 6 章で自然言語処理の出力をもとにした, 仕様書の命令の抽出の方法について述べる. 最後に, 7 章で抽出し形式化した命令をもとに字句解析をするインタプリタの実装について述べ, 8 章で字句解析インタプリタに対してテストを行い, 抽出した命令の正しさを検証した.

第 2 章

準備

2.1　自然言語処理

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で処理する技術を自然言語処理（Natural Language Processing）と呼んでいる. 自然言語処理をすることによって, 自然言語の文章を構成している単語の解析 (単語の品詞や、原型の取得など) や文章の構造の解析, 意味の解析をすることが出来る. このセクションでは, 自然言語処理によって得られるそれらの情報に関しての説明をしていく.

2.1.1　トークン分割, 品詞タグ付け, レンマ化

トークン化では, 文章を単語という単位に分割する操作である. そして品詞タグ付けは, 文章をトークン化した後, その単語の品詞を調べる操作である. 例えば, 品詞には以下のような種類があり, 単語ごとに品詞が定められる.

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
PRP　: 人称代名詞

そして, レンマ化とはその単語の原型を調べる操作である. この操作により, 例えば三人称単数現在形の動詞の原型を調べられる.

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.

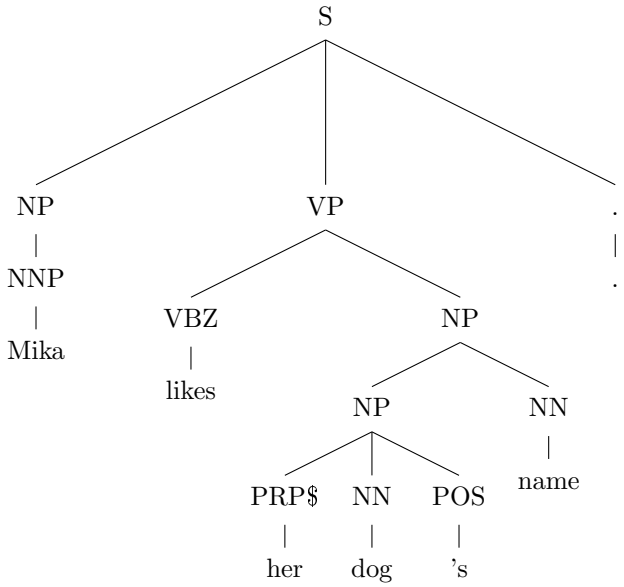
2.1.2 構文木解析

自然言語の文章には, 節や句などのまとまりが存在する. 節は主語と述語がある文のことで, 句は 2 つ以上の単語によって構成されている単語の集まりである. (本論文では節と句は特に区別しない.) 例えば, 節や句の種類には以下のようなものがある.

S : 文
VP : 動詞句
NP : 名詞句
PP : 前置詞句

そして構文木解析とは, 単語を節や句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.

2.1.3 係り受け解析 (dependency parse)

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 という風に添え字付きの矢印で表記する.

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.

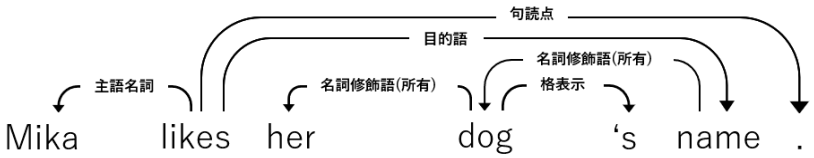


図 2.1　係り受け解析

2.1.4　固有表現抽出 (named entity recognition)

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.

2.1.5　参照関係の解析 (coreference)

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.

2.2　自然言語処理のライブラリの使用

本研究では, 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 を使用した. その使用方法について説明をしていく.

2.2.1　a

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 [7] から得られ, 全て自然言語（英語）によって書かれている.

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, ほとんどの状態は, 文字マッチ前の処理と文字マッチングの処理 (文字とその文字にマッチした時の処理) の要素から成り立っている. 表 3.1 は字句解析器の状態の 1 つである, Character reference state の例である.

表 3.1 HTML5 字句解析仕様の状態：Character reference state

状態名	Character reference state	
文字マッチ前の処理	Set the temporary buffer to the empty string. Append a U+0026 AMPERSAND (&) character to the temporary buffer. Consume the next input character:	
文字マッチ処理	文字	処理
	ASCII alphanumeric	Reconsume in the named character reference state.
	U+0023 NUMBER SIGN (#)	Append the current input character to the temporary buffer. Switch to the numeric character reference state.
	Anything else	Flush code points consumed as a character reference. Reconsume in the return state.

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

命令の形式

BNF の記法を用いて, HTML5 の字句解析の仕様書から抽出する命令を形式化する.

4.1 抽出する命令の形式

まず形式化する命令の, 型とそのメタ変数を以下で定める.

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

CommandValue 型, InplementVariable 型はそれぞれ字句解析器の仕様書に変数や値として出てくる単語を形式化したものであり, InplementVariable 型は値を代入される変数として, CommandValue 型は単に値をとして区別している.
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 $cList_2$ ($cList_1$, $cList_2$ は Command 型のリストの値)
|	Ignore()	// 何もしない
|	Switch(cval)	// cval が状態を表す値の時, 状態 cval へ遷移する
|	Reconsume(cval)	// cval が状態を表す値の時, 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)	// 変数 ival に値 cval を代入する (ival ← cval)
|	AppendTo(cval, ival)	// ival, cval が文字, 文字列を表す値の時, 変数 ival の値に, 値 cval を後ろから追加する (ival ← ival + cval)
|	Emit(cval)	// cval がトークンを表す値の時, トークン cval を排出する
|	Create(string, cval)	// cval がトークンを表す値の時, 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)	// cval が文字, 文字列を表す値の時, 文字 cval を入力文字列から消費する
|	Error(string)	// 構文エラー string を排出する
|	FlushCodePoint()	// 一時バッファの内容を排出する
|	StartAttribute()	// 現在のタグトークンに対して新しい属性を追加する
|	TreatAsAnythingElse()	// 現在の状態の AnythingElse に記述してある処理を実行する
|	AddTo(cval, ival)	// ival, cval が数を表す値の時, 変数 ival の値に, 値 cval を足す (ival ← ival + cval)
|	MultiplyBy(ival, cval)	// ival, cval が数を表す値の時, 変数 ival の値に, 値 cval をかける (ival ← ival × cval)

Bool 型

b ::=	CurrentEndTagIsAppropriate()	// EndTagToken が適切なものである
|	CharacterReferenceConsumedAsAttributeVal()	// CharacterReferenceCode が属性の値として 消費されている
|	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
        |   On                         // フラグの状態：On
        |   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
```

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

4.2 命令形式の例

自然言語の文から, 上記の命令の形式への変換の例をいくつか示す.

状態 Data_state へ遷移するという意味の文 “Switch to the Data_state.” は, Switch(StateName(Data_state)) として抽出される.
(状態名を意味する StateName(Data_state) という CommandValue 型の値を持つ, Command 型の値 Switch)

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 “Append the lowercase version of the current input character to the current tag token’s tag name.” は, Append(LowerCase(CurrentInputCharacter), INameOf(CurrentTagToken)) として抽出される.
(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 型の値 Append)

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

HTML5 字句解析仕様の自然言語処理

HTML5 の字句解析仕様に対して自然言語処理を適用する．そのための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 [4] を使用し, 処理を行った.

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, 仕様書の書かれている構造を認識し, 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　仕様書の構造の認識, 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 [3] 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, “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.

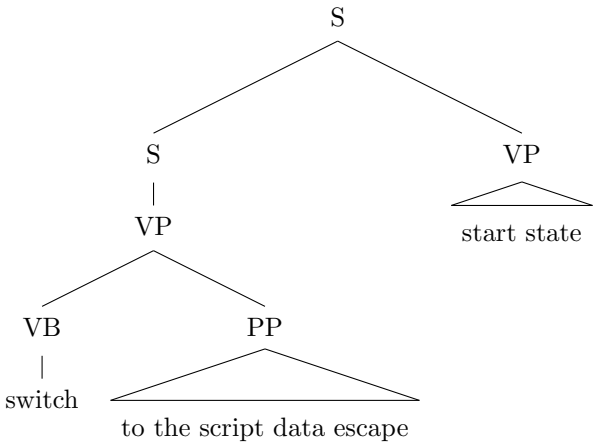
自然言語処理は,「文字マッチングする前の処理」と「文字 i にマッチした時の処理」の部分に対して行う.

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 型の定義を示す.
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

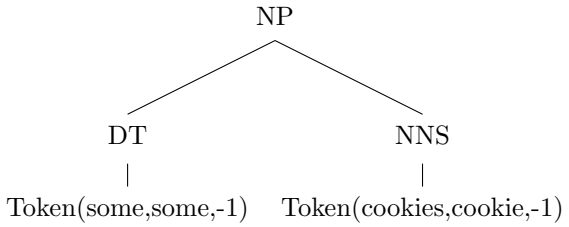
```
trait Tag
case class Node(node: NodeType, list: List[Tag]) extends Tag
case class Leaf(leaf: LeafType, token: Token) extends Tag
case class Token(word: String, lemma: String, coref: Int) extends Tag
trait NodeType
case object S extends NodeType
case object NP extends NodeType
case object VP extends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

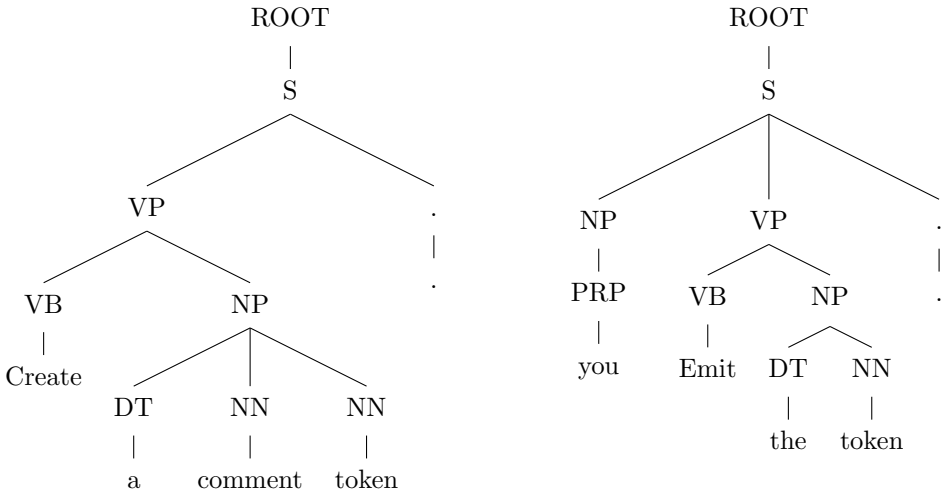
Leaf 型が持つ Token に関しては, 単語の原型の部分にレンマ化の処理で得た原型の情報を代入し, 参照番号の部分で下記の参照関係の処理を行い, 値を代入する.

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.

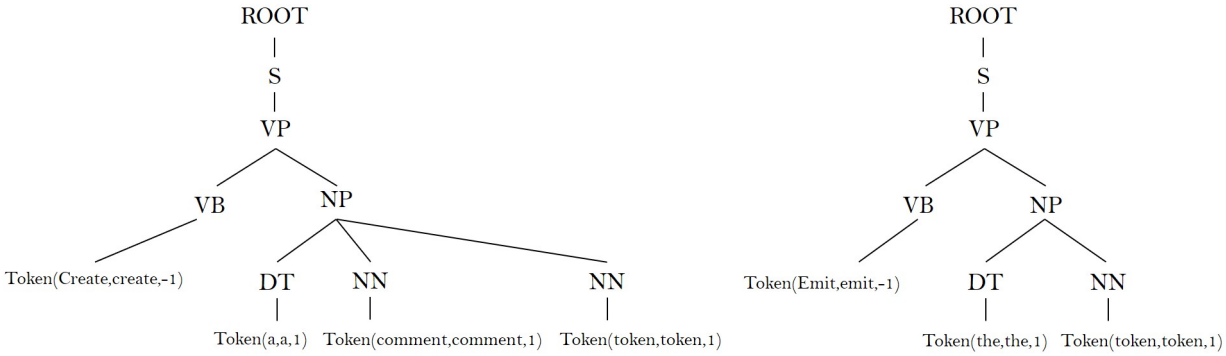


図 5.1　Tag 型

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

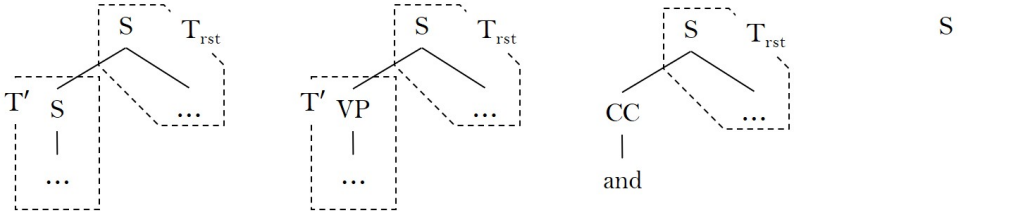
これらを順に説明していく. 木の形のマッチに関しては, 葉の値は単語の原型の情報のみ表記, 使用する.

6.1.1 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,

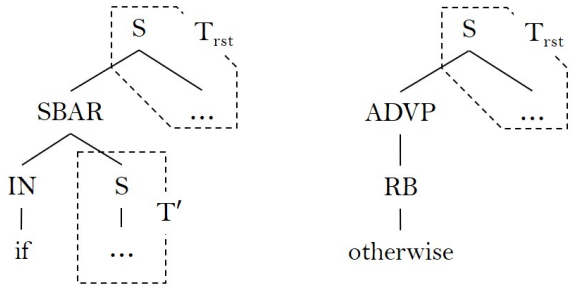


(i) 先頭が S　(ii) 先頭が VP　(iii) 先頭が繋ぎの単語　(iv) 子ノードが空

(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,

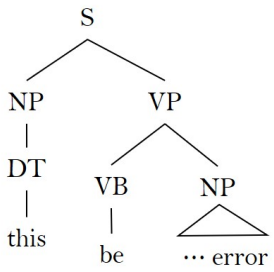


(i)　(ii)

(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

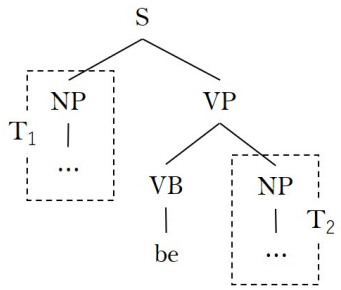
Tag 型の値 T の形が,



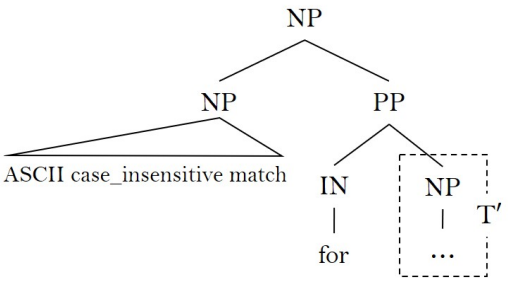
の時, [Error(··· error)] を返す.

6.1.2 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.



部分木 T_1 内の限定詞を除いた文字列が “current end tag token”, 部分木 T_2 内の限定詞を除いた文字列が “appropriate end tag token” の時, CurrentEndTagIsAppropriate()
T_2 の形が,



の場合, AsciiCaseInsensitiveMatch($\mathcal{T}_{NP_C}(\mathrm{T}_1)$, $\mathcal{T}_{NP_C}(\mathrm{T}'_2)$)
それ以外の時, IsEqual($\mathcal{T}_{NP_C}(\mathrm{T}_1)$, $\mathcal{T}_{NP_C}(\mathrm{T}_2)$)

また, 木 T 内にある文字列が の文字列が, “character reference was consumed as part of an attribute” の時, CharacterReferenceConsumedAsAttributeVal() を返す.

6.1.3　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.

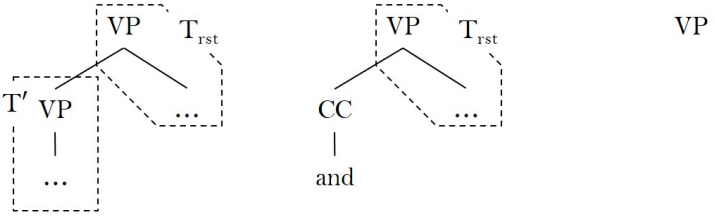
ここで表記していない, 残りの Command 型の命令に関しても, その命令文の構文木の形を調べて, 変換の規則を作っていく.

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)　(ii)　(iii)

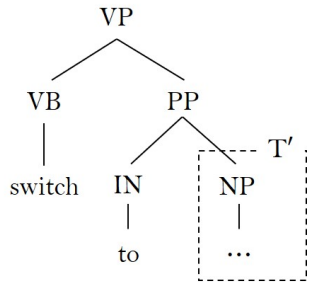
(i) の場合, $\mathcal{T}_{VP}(\mathrm{T}') +\!+ \mathcal{T}_{VP}(\mathrm{T}_{rst})$ を返す.

(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

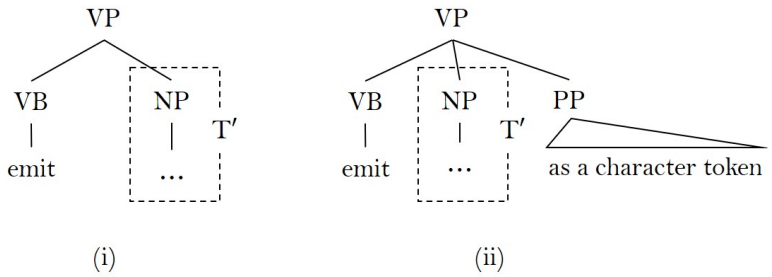
命令文の構文木解析の形を調べ, それに基づいた Tag 型の値のパターンマッチを作る. Tag の形が,



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

Emit 文のマッチ

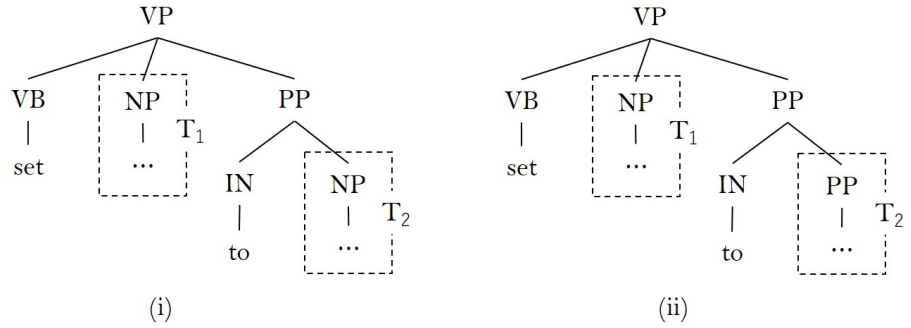
Tag 型の値 T の木構造が,



(i) 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.
(ii) の場合, $\mathcal{D}(\mathrm{T}') = [\mathrm{T}'_1, \ldots, \mathrm{T}'_n]$ 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

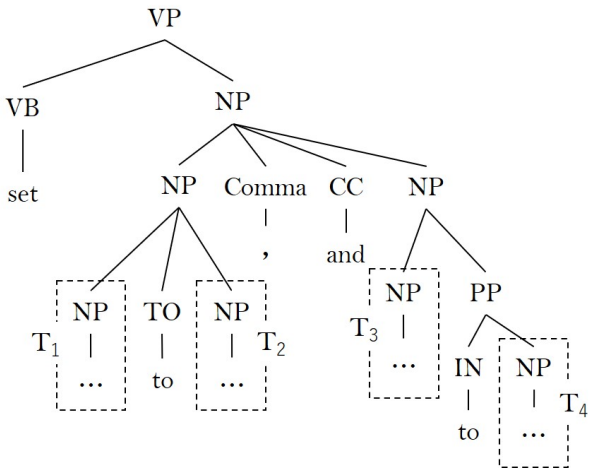
木構造が,



(i) の場合, 変数に値を代入するという意味の Set 文である.
$\mathcal{D}(\mathrm{T}_1) = [\mathrm{T}_1^1, \ldots, \mathrm{T}_1^n]$ の時, [Set($\mathcal{T}_{NP_I}(\mathrm{T}_1^1)$, $\mathcal{T}_{NP_C}(\mathrm{T}_2)$), …,Set($\mathcal{T}_{NP_I}(\mathrm{T}_1^n)$, $\mathcal{T}_{NP_C}(\mathrm{T}_2)$)] を返す.
(ii) の場合, 変数の状態を変えるという意味の Set 文である.
部分木 T_2 内にある文字列が “on” だったら, $\mathcal{D}(\mathrm{T}_1) = [\mathrm{T}_1^1, \ldots, \mathrm{T}_1^n]$ の時, [Set($\mathcal{T}_{NP_I}(\mathrm{T}_1^1)$, On), …,Set($\mathcal{T}_{NP_I}(\mathrm{T}_1^n)$, On)] を返す.

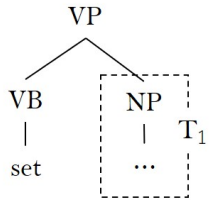
個別に対応した Set 文のマッチ

“Set that attribute’s name to the current input character, and its value to the empty string.” のような, 1 文の中に 2 つの命令が含まれている文章も仕様書内に出てくるが, このような文の構文木は,



のように, 特殊な形になっている. よってこのような木構造になっているものの Command 型の変換を個別に対応し, [Set($\mathcal{T}_{NP}$(T$_1$), $\mathcal{T}_{NP}$(T$_2$)), Set($\mathcal{T}_{NP}$(T$_3$), $\mathcal{T}_{NP}$(T$_4$))] を返すようにした.

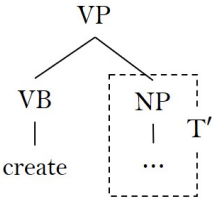
また, “Set the self-closing flag of the current tag token.” のような状態の切り替え先を示す “to on” が省略されている文章も仕様書内に存在する. このような文に対応するため, Tag 型の値 T が,



にマッチした場合は, [Set($\mathcal{T}_{NP}$(Node(NP, np)), On)] を返すようにした.

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(“n”, $\mathcal{T}_{NP}$(T′))] を返す.

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

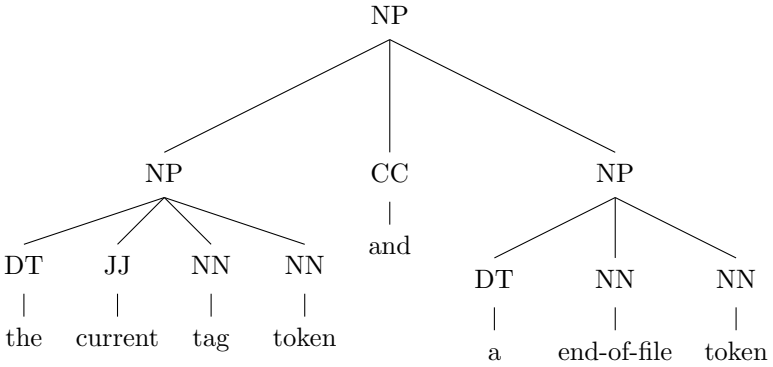
6.1.4　名詞句 (NP ノード) の分割 $\mathcal{D}$

例えば, “Emit the current tag token and an end-of-file token” という文を, 形式化する際, Emit(“the current tag token and an end-of-file token”) みたいな感じではなく, Emit(“the current tag token”) and Emit(“an end-of-file token”) の様にしたい. よって, そのようなことが出来るように, 名詞を分割し, それをリスト化する関数を作成した.

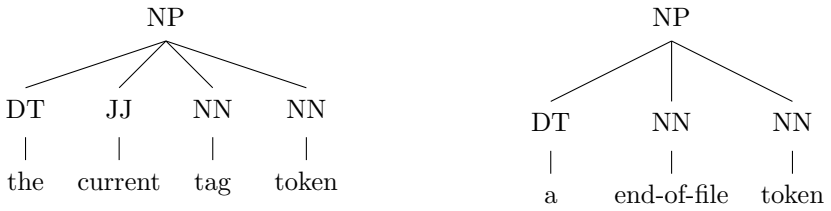
$\mathcal{D}$ は Tag 型の値 T を受け取り, それを分割した Tag 型のリストの値を返す.

名詞句には, “A and B” や “A, B” のような, 複数の名詞句の連結の形であることがある. $\mathcal{D}$ はそれらを分解し, Tag 型のリストとして出力するものである. (このような連結の形でないときは, 単元リスト [T] を返す.)

例えば, 以下の and で繋がれている名詞句を “the current tag token and an end-of-file token”



以下の 2 つの名詞句の Tag 型に分解する.



6.1.5　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. $\mathcal{T}_{NP_C}$ と $\mathcal{T}_{NP_I}$ は変換後の型が違うのみで, 同じ動作をする.

Tag 型の値 T から, CommandValue 型や ImplementVariable 型へ変換する際, T 内の文字列に変数名や値などを表す特定の文字列が含まれているかどうかと, 何らかの参照関係が存在するかを調べるというやり方をとった.

例えば, T 内に”return state”という文字列が含まれていた場合は, $\mathcal{T}_{NP_C}$ の場合は ReturnState, $\mathcal{T}_{NP_I}$ の場合は IReturnState を返す.

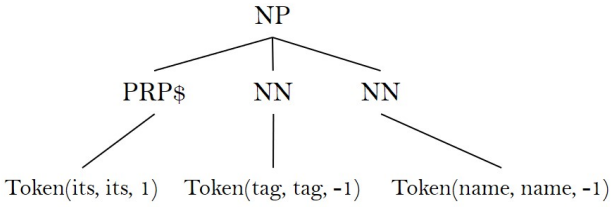
ある CommandValue 型の値 cval に変換され, 参照番号が-1 でない数字 n のとき (その単語にラベル n の参照関係があるとき) は, 字句解析器の実行において, CommandValue 型の意味を解釈する際, 変数”n”に解釈後

の cval の値を代入するという意味を持つ, Substitute(“n”, cval) 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 “n” という意味を持つ, Variable(“n”) を返す. また, name, value が含まれており, 参照番号が-1 でない数字 n の場合は NameOf(Variable(“n”)), ValueOf(Variable(“n”)) を返す.

例

Tag 型の値が



の場合, この文字列には, name という単語が含まれているかつ, “its” に参照関係の番号 1 がついているので, NameOf(Variable(“1”)) を返す.

6.2 条件分岐文の処理

Command 型への変換の時に, If(〈 条件文 〉, 〈True の時の処理 〉, 〈False の時の処理 〉) の様に, If 文の内容をまとめて取り出すのではなく, 条件文節は, それを If_(〈 条件文 〉) という風に単体で取り出し, Otherwise という単語もそれ単体で Otherwise_() という風に取り出したのは, 単体で取り出したものの列を, Otherwise の処理がどこまでなのかの判別が難しいという理由で, If(〈 条件文 〉, 〈True の時の処理 〉, 〈False の時の処理 〉) という形に, 手動で組み立てようと考えからである.

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もあった. 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.

処理内容

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

[If_(b), command_1, ... , command_m, Otherwise_(), command_{m+1}, ... , command_n]

のような形があったら, これを以下の考えられる If 文に組み立て, その中から正しいほうを人の手で選ぶ.

[If(b, [command_1, ... , command_m], [command_{m+1}, ... , command_n])]

[If(b, [command_1, ... , command_m], [command_{m+1}, ... , command_{n-1}]), command_n]

...

[If(b, [command_1, ... , command_m], [command_{m+1}]), command_{m+2}, ... , command_n]

処理の例

条件文の処理をする前の Command 列
[If_(b), Switch(···), Otherwise_(), Switch(···), Emit(···)] は,
⇒ [If(b, [Switch(···)], [Switch(···), Emit(···)])]
⇒ [If(b, [Switch(···)], [Switch(···)]), Emit(···)]
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.

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)

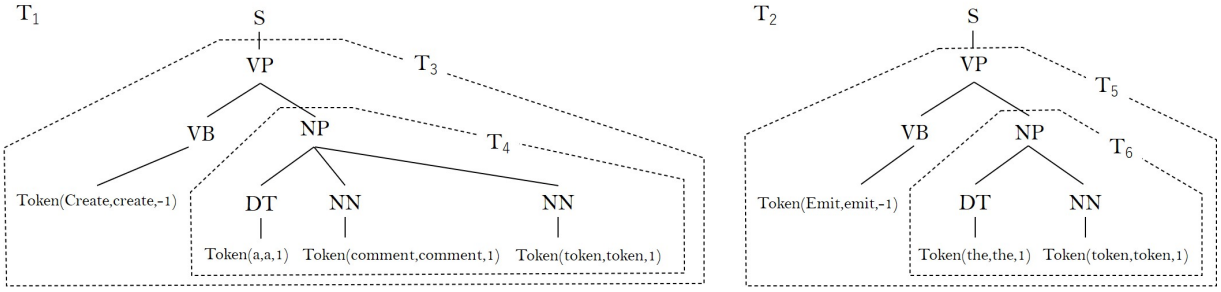


図 6.1 例 5.4 で変換された Tag 型

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

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

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

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 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
)
```

6.4.1 例

表 6.1 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.

表 6.1 の状態に 5, 6 章の処理をし, 1 状態の形式的な定義に変換すると, 図 6.1 のようになる.

Listing 6.1 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 7 章

字句解析器の実装

6 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.

7.1　実装の概観

字句解析インタプリタは, 図 7.1 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.

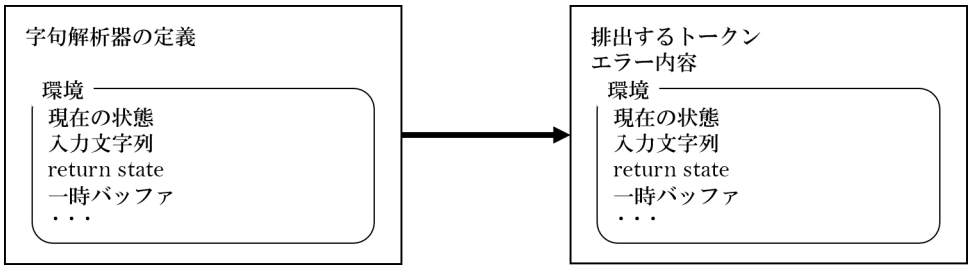


図 7.1　インタープリタ概略

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.

7.1.1　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.

7.2 インタプリタの実装の詳細

7.2.1 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(name: String, public_identifier: String, system_identifier: String, force_quirks_flag: Boolean), 開始タグトークンを表す startTagToken(name: String, self_closing_flag: Boolean, attributes: List[Attribute]), 終了タグトークンを表す endTagToken(name: String, self_closing_flag: Boolean, attributes: List[Attribute]), コメントトークンを表す commentToken(data: String), 文字トークンを表す characterToken(data: String), EOF トークンを表す endOfFileToken() を有する.

タグトークンが持つ属性を表す型 Attribute は Attribute(name: String, value: String) と, 名前, 値を持つクラスである.

7.2.2 字句解析器の変数の型：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.

7.2.3 文字マッチングの処理

字句解析の状態において, 現在消費した文字がどういう種類の文字なのかによって, どの処理をするかの分岐が起こる. その文字マッチングの処理を実装した.

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.

7.2.4 CommandValue 型の解釈

CommandValue 型の値と環境を受け取り, 変更後の環境と Value 型を返す関数を実装した.

処理の例

入力が CommandValue 型 ReturnState, 字句解析の環境 env である場合, env の変数である,“return state” の値を参照し, その値を返す.

入力が Variable(“1”), 字句解析の環境 env である場合, env の変数 “1” を参照し, その値を返す.

入力が Substitute(“1”, CString(“abc”)), 字句解析の環境 env である場合, env に対して 変数 “1” に StringVal(“abc”) を代入し, StringVal(“abc”) を返す.

7.2.5 Command 型の解釈

Command 型の値と環境を受け取り, 変更後の環境, 排出トークン, エラー内容を返す関数を実装した.

処理の例 1

入力が Command 型の値 `Switch(StateName(‘‘Data_state’’))`, 字句解析の環境 env である場合, これは, “Data_state” という名前の状態に遷移するという意味の命令なので, 字句解析の環境 env の “現在の状態” に `StateVal(‘‘Data_state’’)`(StateName(“Data_state”) を CommandValue 型の解釈する関数で Value 型にした値) を代入するという処理を行う.

処理の例 2

入力が Command 型の値 `Set(IReturnState, StateName(‘‘Data_state’’))`, 字句解析の環境 env である場合, これは, 字句解析の変数である “return state” の値を, 状態 “Data_state” にするという意味の命令なので, 字句解析の環境 env の “return state” に `StateVal(‘‘Data_state’’)`(StateName(“Data_state”) を CommandValue 型の解釈する関数で Value 型にした値) を代入するという処理を行う.

7.2.6 Bool 型の解釈

Bool 型の値と環境を受け取り, 変更後の環境と真偽 (Boolean 型の値) を返す関数を実装した.

例えば, CharacterReferenceCode が属性の値として消費されているかという条件文は, 字句解析の環境の return state の値が字句解析の状態である “attribute value (double-quoted) state” か “attribute value (single-quoted) state” か “attribute value (unquoted) state” のいずれかであったら, `true` であるという意味の文である.

よって, 入力としてその意味を持つ Bool 型の値 CurrentEndTagIsAppropriate(), return state が “attribute value (unquoted) state” となっているような字句解析の環境 env がきたら, `true` を返す.

第 8 章

実装の評価

7 章で実装した字句解析インタプリタの関数をそれぞれテストした後, HTML5 の字句解析のテストである, html5lib-tests [1] の tokenizer のテストデータを用い, 6 章で抽出した命令を字句解析インタープリタの定義として入れ, 実装した字句解析器のテストをし, 抽出した命令の正しさを検証した.

8.1 概要

使用した HTML5 字句解析器のテストデータは, 14 つのテストファイルから成り立っており, 1 つのテストファイルには複数のテストが含まれている.

テストファイルには, '&' から始まる HTML 特殊文字コードの処理のテストをする entities.test, 偽のコメントトークンに対する処理のテストをする escapeFlag.test, Named character references 状態における, 表の参照のテストをする namedEntities.test, character reference code の値から文字への参照のテストをする numericEntities.test, DOCTYPE やタグトークンなどのトークンが適切に排出されるかのテストをする contentModelFlags.test, domjs.test, pendingSpecChanges.test, test1-4, ユニコードで書かれた文字列の処理が上手くいくかのテストをする unicodeChars.test, 不適切な場合のユニコードの処理のテストをする unicodeCharsProblem, 不適切な XML のコードの処理のテストである xmlViolation.test がある.

テストは 1 つあたり, プログラム 8.1 のような JSON 形式で記述されており, "description"にそのテストの名前, "initialStates"に字句解析する際の初期状態 (複数ある場合はそれぞれの場合を別々に実行する), "lastStartTag"に直前に排出されている開始タグトークンの名前, "input"に字句解析器への入力文字列, "output"出力すると想定されるトークン列, "errors"に字句解析をする際出力され得る構文エラーの情報を持つ.

Listing 8.1 HTML5lib contentModelFlags.test

```
{
  "description":"End tag closing RCDATA or RAWTEXT (ending with space)",
  "initialStates":["RCDATA state", "RAWTEXT state"],
  "lastStartTag":"xmp",
  "input":"foo</xmp ",
  "output":[["Character", "foo"]],
  "errors":[
      { "code": "eof-in-tag", "line": 1, "col": 10 }
  ]
}
```

本研究で行うテストは, 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. そしてそれらが両方一致していたら成功とする.

HTML5 字句解析テストファイルのうち, “domjs.test” の中の “CDATA in HTML content” という名前のテストと, “xmlViolation.test” のテストは, 字句解析器の命令の実装以外の部分も実装する必要があったので, 今回のテストからは省いた.

8.2 HTML5 字句解析テストの実行

表 8.1 状態名の置き換え, “-” の “_” への置き換えのみをした場合のテスト結果

テストファイル名	結果
contentModelFlags.test	4/24
domjs.test	12/57
entities.test	1/80
escapeFlag.test	0/9
namedEntities.test	2231/4210
numericEntities.test	1/336
pendingSpecChanges.test	0/1
test1.test	24/68
test2.test	17/45
test3.test	757/1786
test4.test	38/85
unicodeChars.test	322/323
unicodeCharsProblem.test	5/5

表 8.2 文字列の置き換えの処理した場合のテスト結果

テストファイル名	結果
contentModelFlags.test	24/24
domjs.test	57/57
entities.test	80/80
escapeFlag.test	9/9
namedEntities.test	4210/4210
numericEntities.test	336/336
pendingSpecChanges.test	1/1
test1.test	68/68
test2.test	34/45
test3.test	1346/1786
test4.test	81/85
unicodeChars.test	323/323
unicodeCharsProblem.test	5/5

状態名の置き換え, “-” の “_” への置き換えのみを文字列の置き換えの前処理として行った場合, 表 8.2 の結果になり, 5 章の置き換えの前処理をすべて行った場合, 表 8.2 の結果になる.

文字列の置き換えの工夫をしなかったら, 適切に命令の抽出が出来ず, 字句解析器のテストを通してもあまり上手くいかなかったが, 置き換えの工夫を行った後は, 字句解析のテストの結果もよくなった.

8.2.1 上手くいかなかった点

文字列の置き換えの処理や, 構文木のアドホックな処理などをした状態でのテストである, 表 8.2 の結果において, 上手くいかなかったテストがある理由として, 適切に命令を抽出できたと見えても, 実際は正しいものから少しずれていたというケースがあった.

例えば表 8.3 の字句解析の状態 “After DOCTYPE name state” において, 入力文字列が “public ···” であるとき, まず文字 ‘p’ を消費し, その文字マッチングにより Anything else の処理を行う. その

処理の文の中の “If the six characters starting from the current input character are an ASCII case-insensitive match for the word “PUBLIC”, then consume those characters” の部分において, これを形式化すると, [If(AsciiCaseInsensitiveMatch(Substitute(“1”, CharactersFromCurrentInputCharacter(6)), CString(“PUBLIC”))), Comsume(Variable(“1”))] となので, この処理を行うとき, 変数 “1” は “public” という文字列を指すことになる. よって Comsume(Variable(“1”)) は文字列 “public” を消費するという操作になる.

しかし, この時入力文字列が “ublic · · ·” という状態であるはずだから, 適切に処理をするために本来は文字列 “ublic” を消費すべきである. このように, 人間なら “those characters” の指しているものをそのまま消費すると上手くいかないとわかることを, 機械だとうまく認識できないという問題が見られた.

表 8.3　HTML5 字句解析仕様の状態： After DOCTYPE name state

状態名	After DOCTYPE name 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
	· · ·	· · ·
	Anything else	If the six characters starting from the current input character are an ASCII case-insensitive match for the word ”PUBLIC”, then consume those characters and switch to the after DOCTYPE public keyword state. Otherwise, · · ·

この問題をアドホックな形で手動で解決させた結果, 以下の表 8.2.1 のテスト結果になり, テストの内容がすべて成功していたので, 仕様書からの命令の抽出が基本的には, 上手くいったと思われる.

表 8.4　処理した場合のテスト結果

テストファイル名	結果
contentModelFlags.test	24/24
domjs.test	57/57
entities.test	80/80
escapeFlag.test	9/9
namedEntities.test	4210/4210
numericEntities.test	336/336
pendingSpecChanges.test	1/1
test1.test	68/68
test2.test	45/45
test3.test	1786/1786
test4.test	85/85
unicodeChars.test	323/323
unicodeCharsProblem.test	5/5

第 9 章

結論

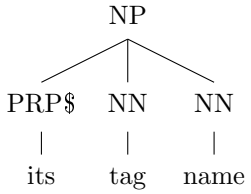
本論文では, HTML5 の字句解析仕様に対して, その命令の形式化をし, 自然言語処理を適用をし, 自然言語処理から得られる情報のうち, 主に, 構文木解析と参照関係の解析を用いることで, 命令の自動形式化を行った. そして, 字句解析のインタプリタを作成し, HTML5 の字句解析のテストをして, 仕様書に書いてある通りの意味の命令を抽出出来たことを確認できた.

しかし, 構文木から命令型への変換において, 構文木の形が標準の形と違ってしまう文に対して, 個別に対応する必要あり, 名詞句の Tag 型の値から CommandValue 型への変換が単に文字列のマッチングをするという愚直なやり方になってしまった部分がある.

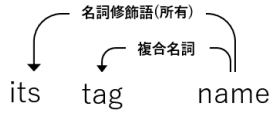
その解決策として, 構文木解析の情報のみではなく係り受け解析の情報も利用する方法があると思われる.

例えば, Tag 型から命令の型である Command 型へ変換する際, 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きた. 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, 名詞句の Tag 型の値から 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, 名詞句を解析する際, 構文木解析と係り受け解析併用することを検討したい.

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] J. G. Greghi, E. Martins, and A. M. B. R. Carvalho. Semi-automatic generation of extended finite state machines from natural language standard documents. In *2015 IEEE International Conference on Dependable Systems and Networks Workshops*, pages 45–50, 2015.

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

[4] 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.

[5] Yasuhiko Minamide and Shunsuke Mori. Reachability analysis of the HTML5 parser specification and its application to compatibility testing. *FM 2012: Formal Methods*, 2012.

[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.