

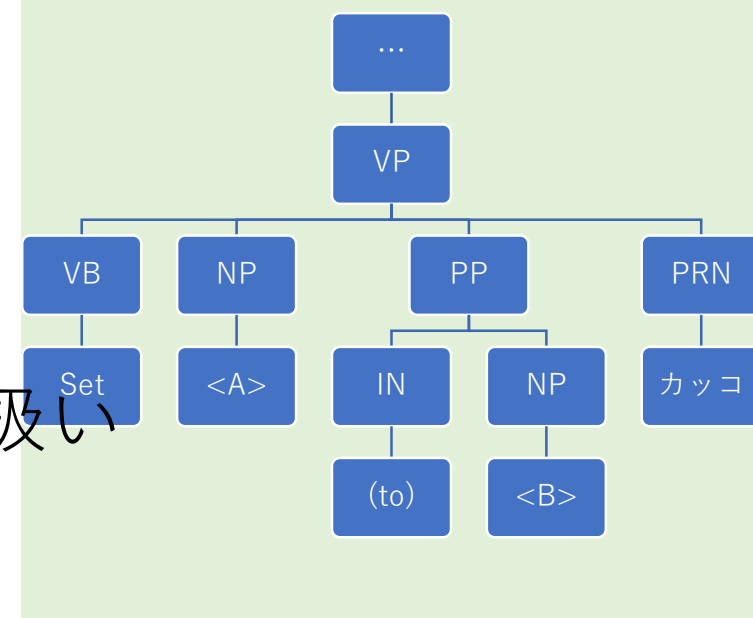
概要など

# やったこと

- 解釈がおかしくなる文について調べた
- 参照関係
- Command変換の図をつかった

# 課題

- Start,treatといった1パターンしかない命令の扱い
- カッコの中身の処理
- 参照関係
- 代名詞を取り除く
- Switch,reconsumeがなかったら、今いる状態へswitchするようになる
- Np-tmp



# カッコの中身の処理をどうするか

Append the lowercase version of the current input character (add 0x0020 to the character's code point) to the current tag token's tag name.

Emit a U+003C LESS-THAN SIGN character token, a U+002F SOLIDUS character token, and a character token for each of the characters in the temporary buffer (in the order they were added to the buffer).

= > 無視するかどうか？

# 参照関係

Create a new end tag token, set its tag name to the empty string.

= > ×

Append the lowercase version of the current input character (add 0x0020 to the character's code point) to the temporary buffer.

= > ○

Create a new DOCTYPE token. Set the token's name to a U+FFFD REPLACEMENT CHARACTER character. = > ○

Create a new DOCTYPE token. Set its force-quirks flag to on. Switch to the data state. Emit the token. = > △

➡ 文と文を & で繋ぐとうまくいく.

“,”と”.”を” and”で置き換える? (htmlPraseの後)

“, and”

# 文の解釈がおかしい(memo)

Emit文の解析結果がおかしい(例外：6-EOF)=>U+ 0 0 3 Cを1つのトークンとして認識させる  
=>”.”の問題は”.”を”.”に置き換えすれば解決するかも.

9-SOLIDUS,19-(-)のswitch文の解釈がおかしい

Reconsume,multiply,flushの解析がおかしい.(命令文の解析が苦手そう、youなどの主語を最初につければ正しく解釈される)

Flush code points consumed as a character reference.

Add a numeric version of the current input character as a hexadecimal digit (subtract 0x0037 from the character's code point) to the character reference code.

Set the return state to the attribute value (double-quoted) state.

=>カッコを',”に変えたらいい感じにできた

# 原因、解決策

StanfordCoreNLPは命令文の解釈が苦手らしい。=>文頭にyouなどの主語を加えて解析させると上手く解釈してくれるようになる。

(解決策) (案)

=>多少無理やりだが、前処理として命令文の文頭にyouを加えて自然言語解析させて、PraseTree->Tagの処理時、加えた"you"の部分のNodeを取り除けば出来そう。

(他：命令文の解釈に強いNLPを使う(なにがいいのか、実際に強いのかわからない)、NLPをトレーニングさせる(難しそう))

openNLPはそのままだと微妙。“you”を加えるといい感じ。Set the return state to the attribute value (double-quoted) state.ができる。

“you”の挿入方法・・・命令文の動詞の数は限られている(13個)。単語でマッチさせて、マッチした単語の前に“you”を加える。(タイミング：htmlPraseの後) (応急処置的)  
(Set the return state to the attribute value (double-quoted) state.が無理.)

# おかしい解釈の詳細

- Reconsume in …  
=>”reconsume”が辞書に載ってないから動詞だと認識されない.
- Emit a U+003E GREATER-THAN SIGN character token.
- Append two U+002D HYPHEN-MINUS characters (-) to the comment token's data.  
=> “U+003E”が一つの単語認識されないから変に解釈される影響
- Switch to the script data escaped dash dash state.  
=> “escaped”が文全体の動詞だと認識されてしまう.
- Multiply the character reference code by 16.  
=> “multiply”が副詞認識される
- Flush code points consumed as a character reference.  
=> “points”が動詞だと思われる
- Set the return state to the attribute value (double-quoted) state.  
=>カッコのせいでややこしくなるから
- Append two U+002D HYPHEN-MINUS characters (-) and a U+0021 EXCLAMATION MARK character (!) to the comment token's data.  
=>!が文末の記号扱いされるので2つの文に分割される=>!を何かで置き換える



# 未対応の文

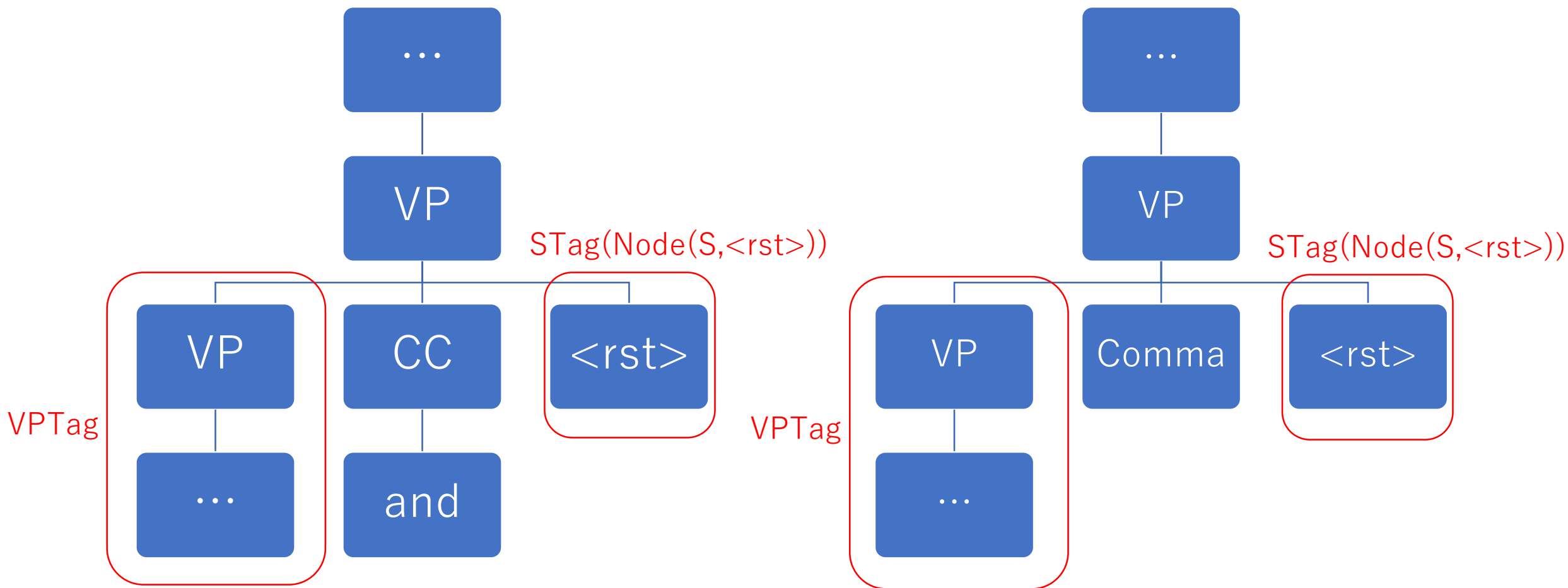
- Set the DOCTYPE token's public identifier to the empty string (not missing), then switch to the DOCTYPE public identifier (double-quoted) state.

=>

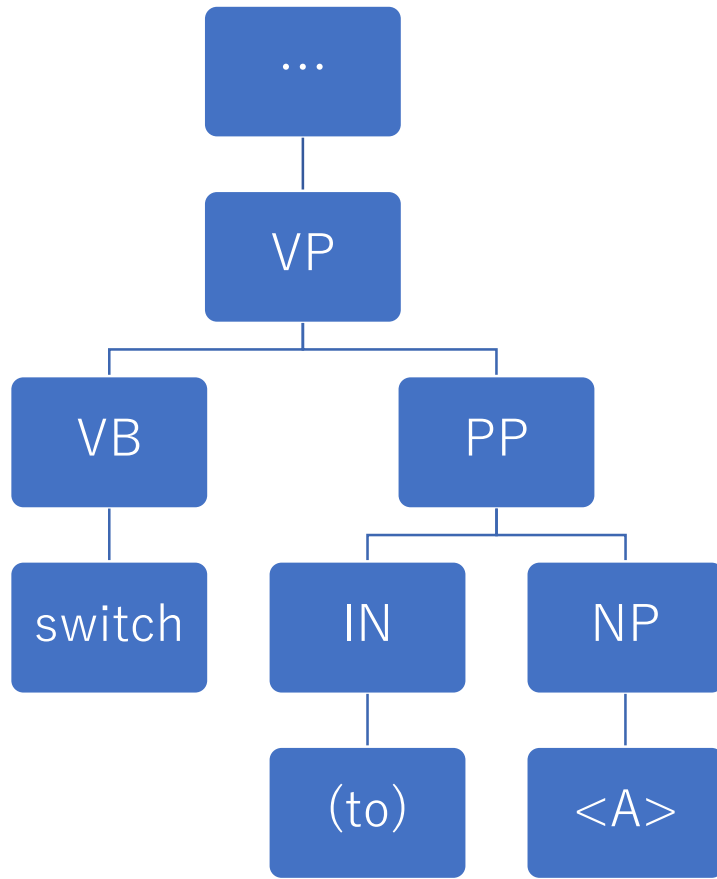
- Otherwise, if(56) => “,”と”.”を” and”で置き換えるとうまく出来そう.

Tree - VPTag

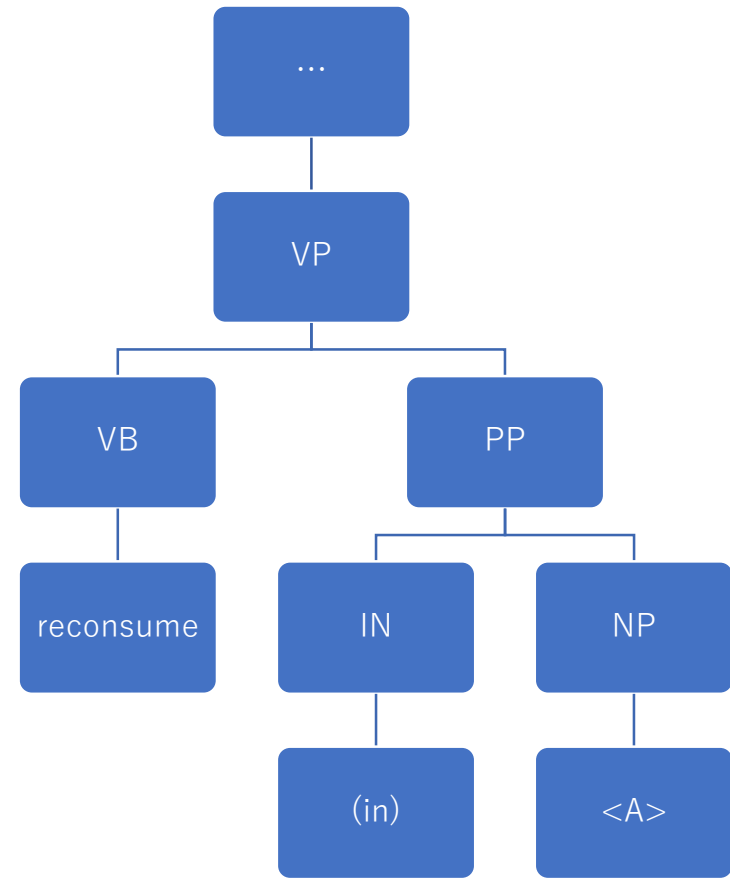
& , " "



# switch, reconsume

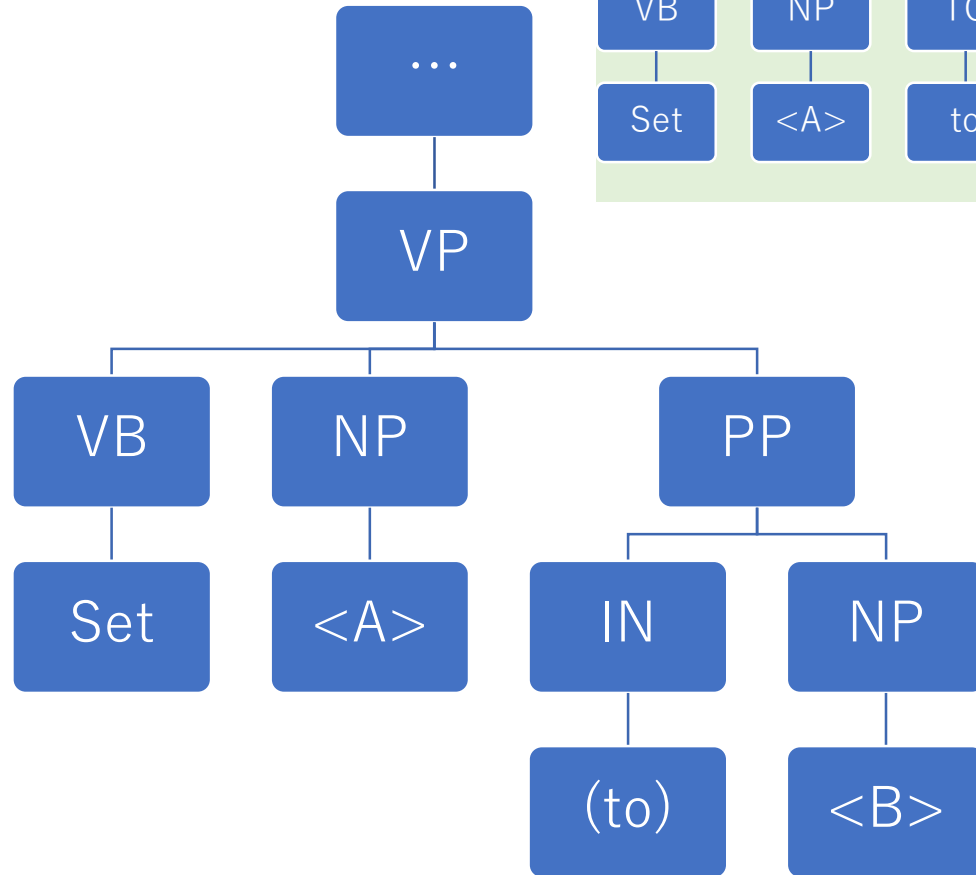
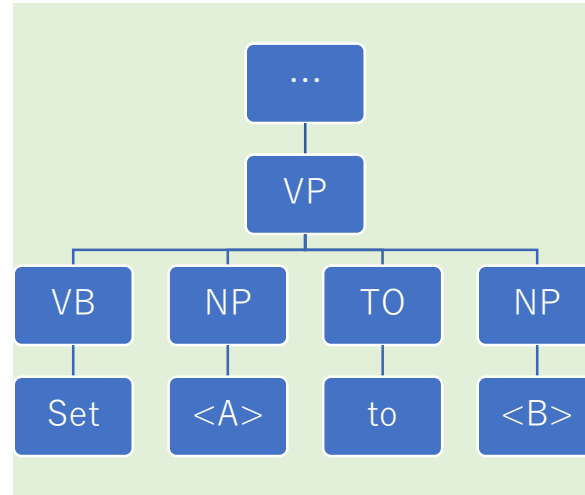


Switch(<A>)

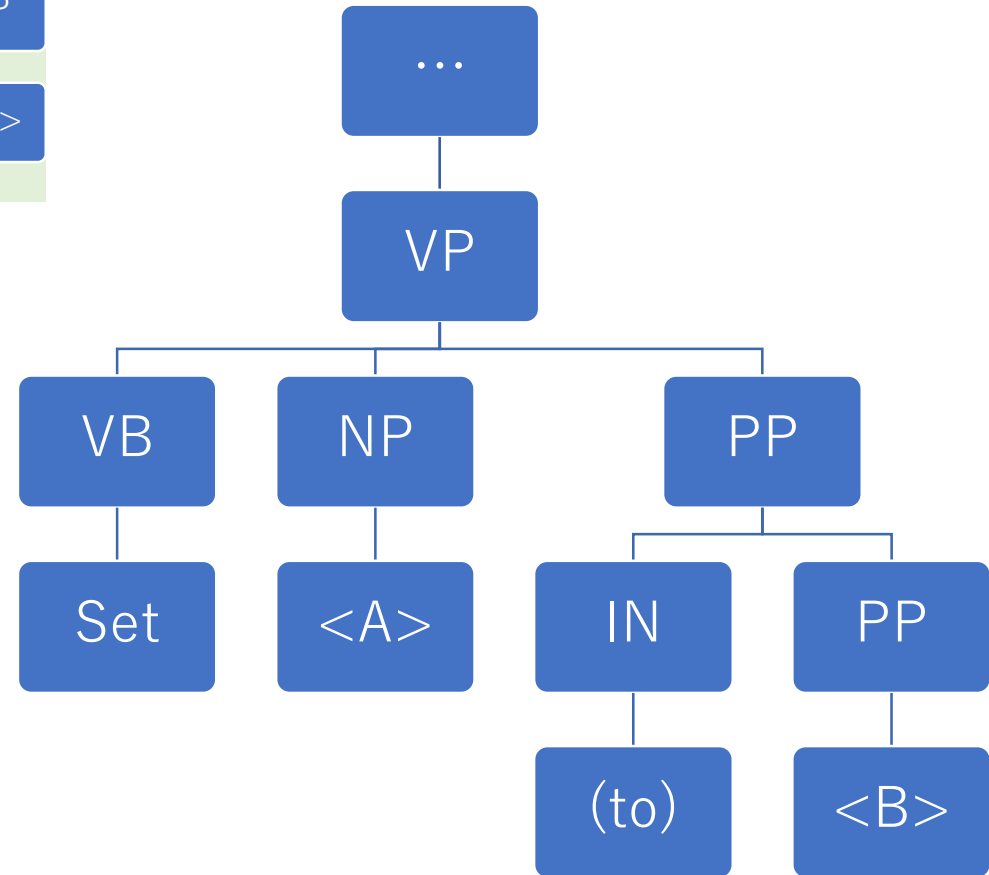


Reconsume(<A>)

set

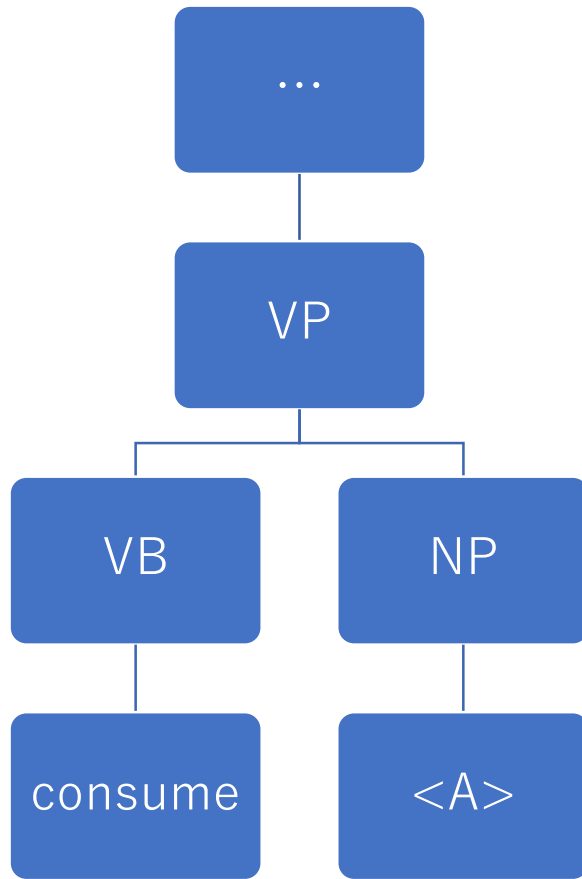


Set(<A>, <B>)

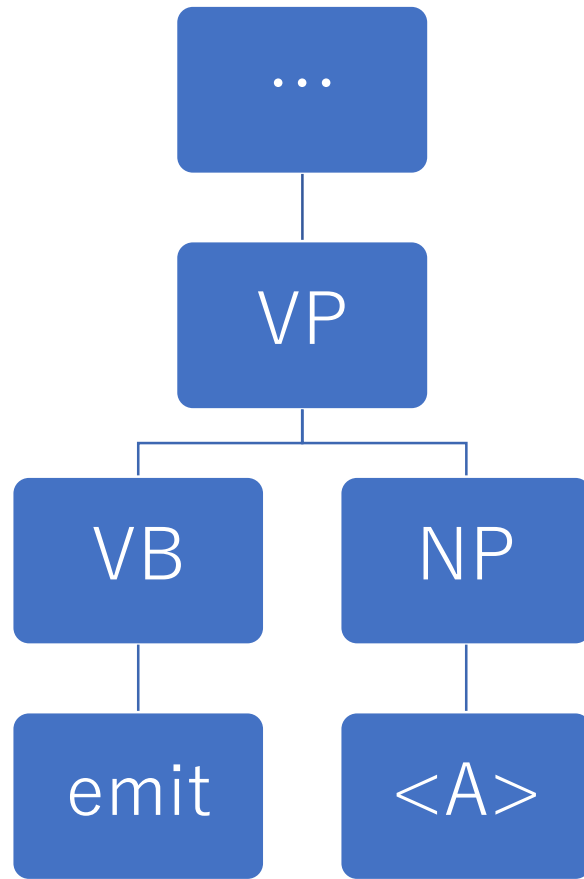


Set(<A>, <B>)

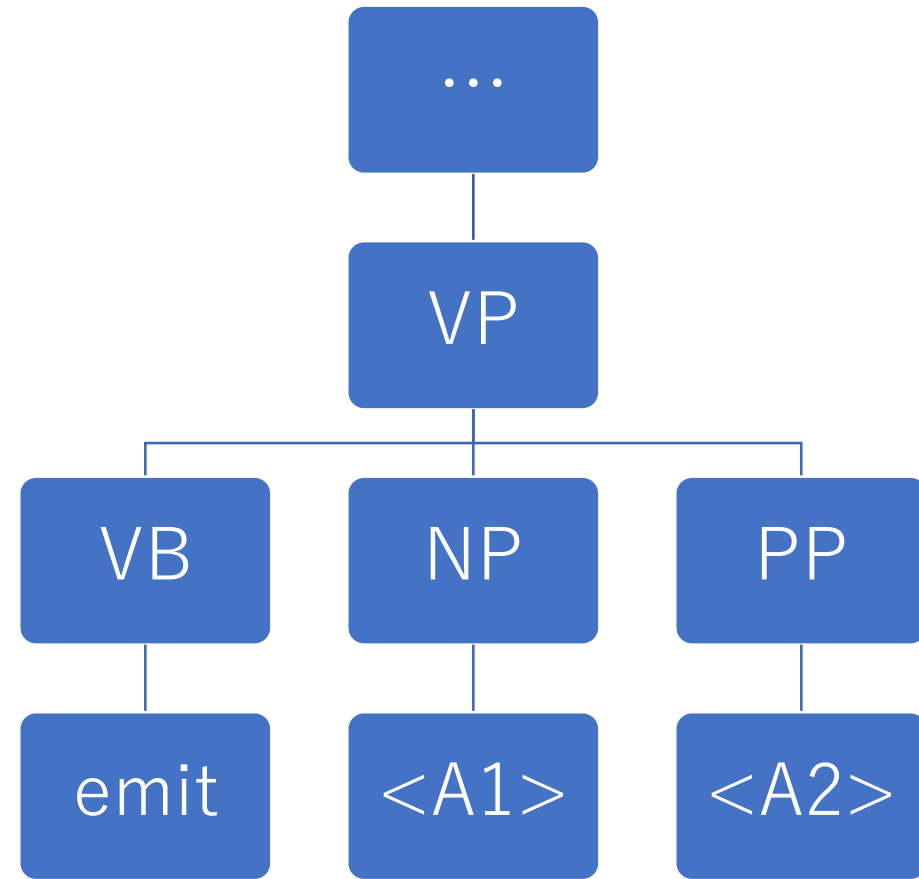
consume,emit



Consume(<A>)

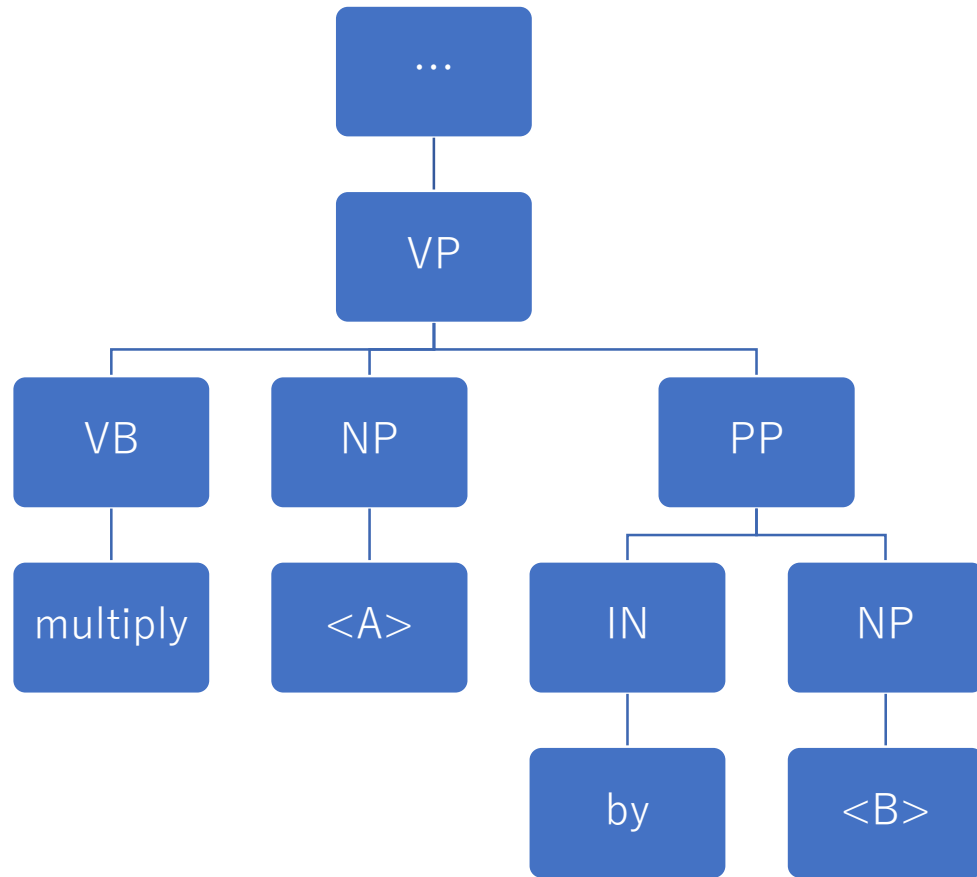


Emit(<A>)

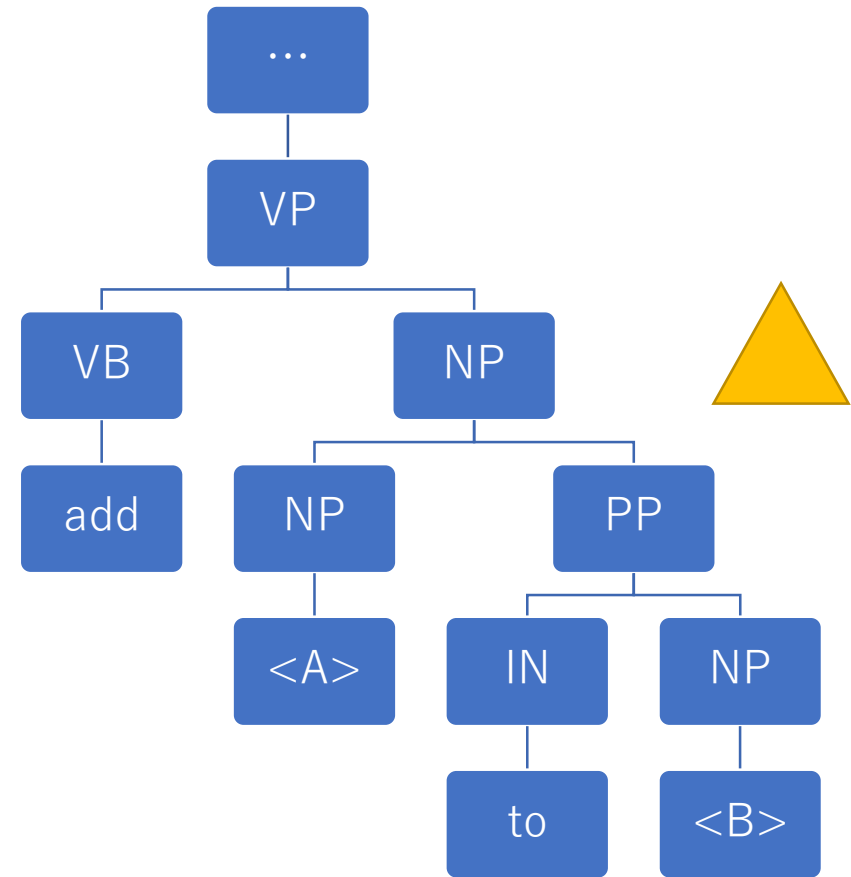


Emit(<A1><A2>)

# Multiply, add

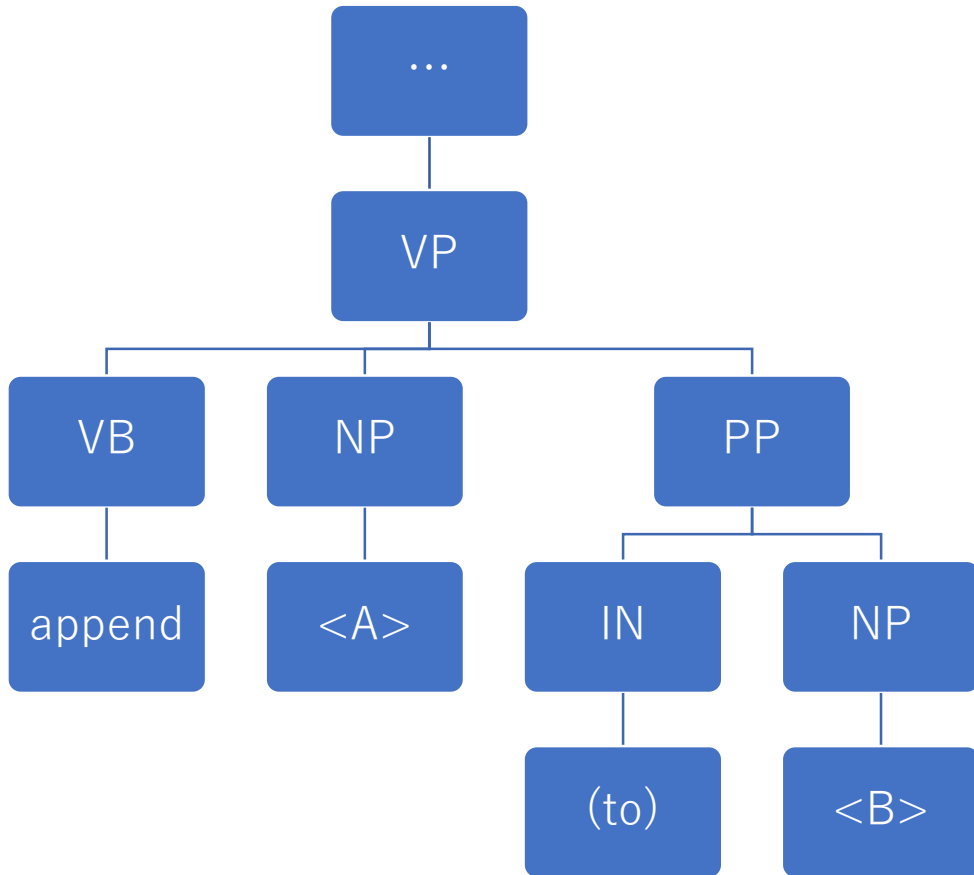


Multiply(<A>, <B>)

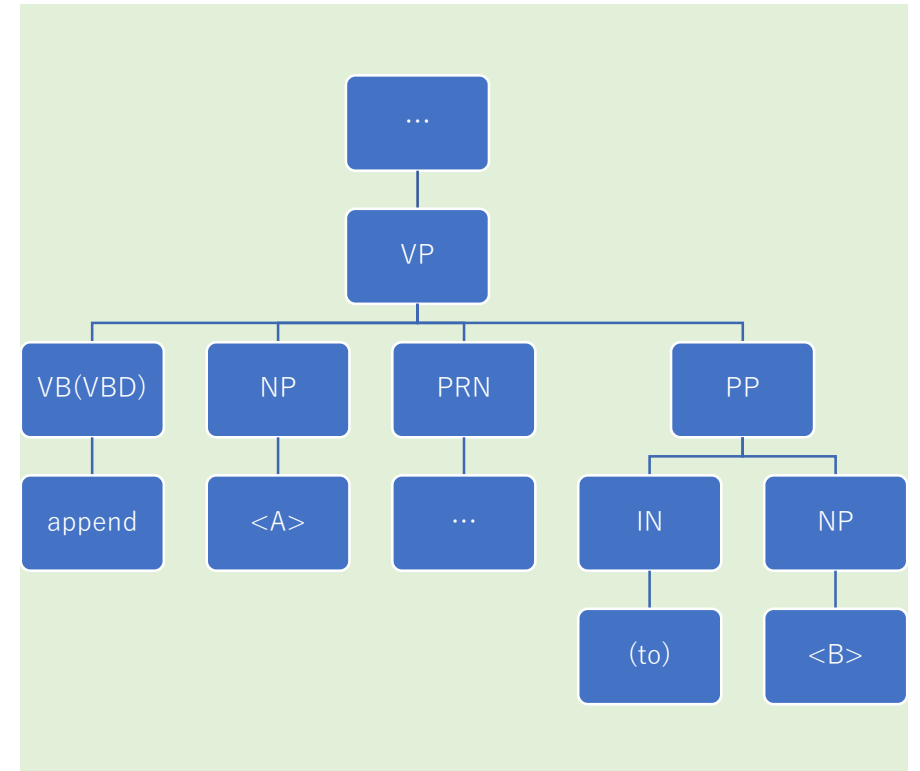


Add(<A>, <B>)

# Append

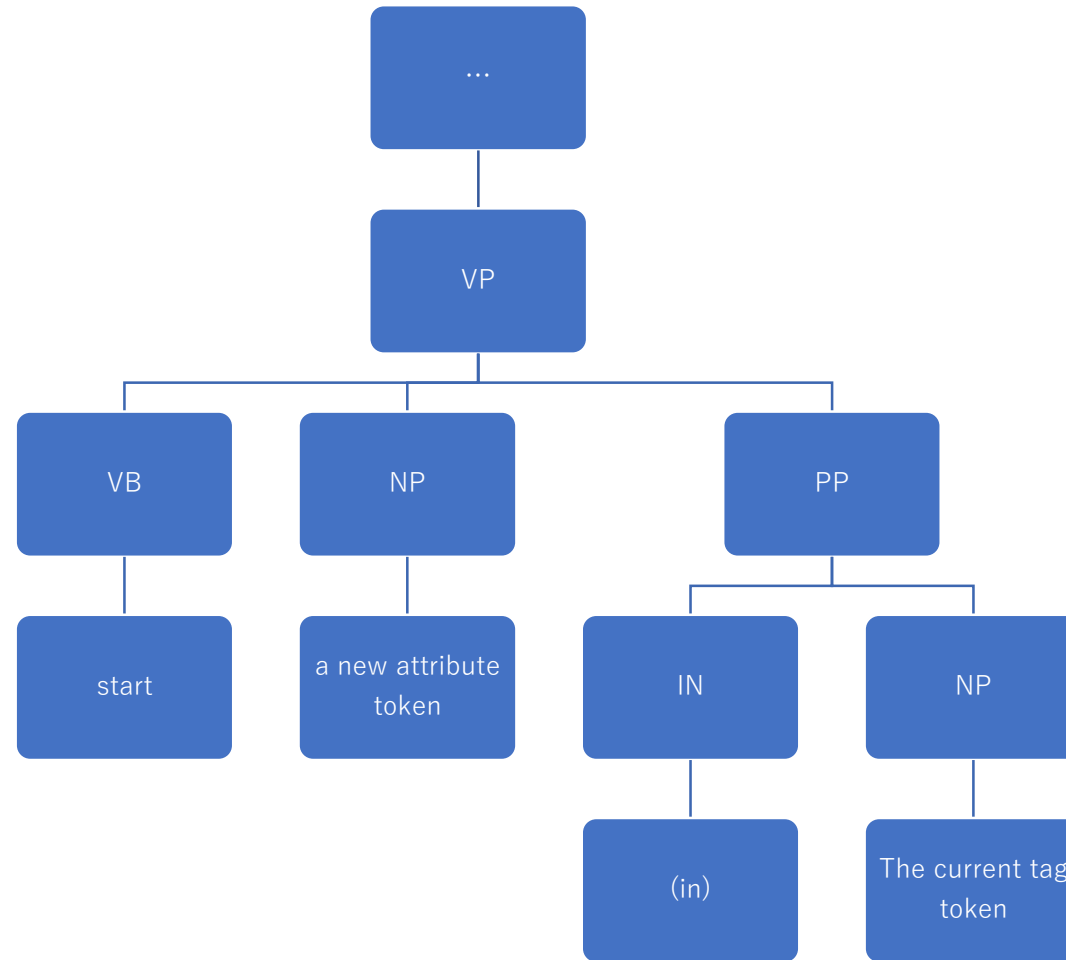


Append(<A>, <B>)



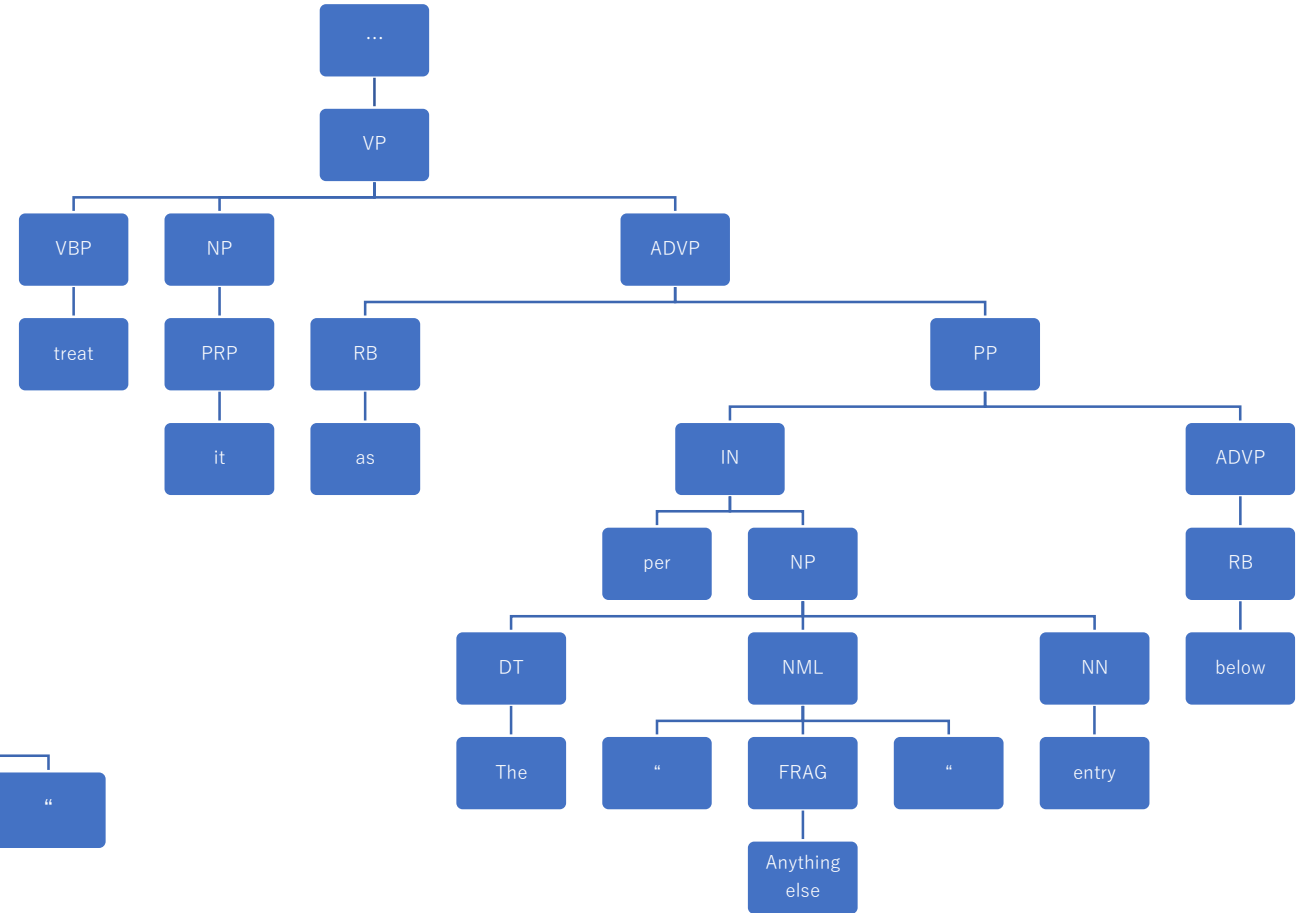
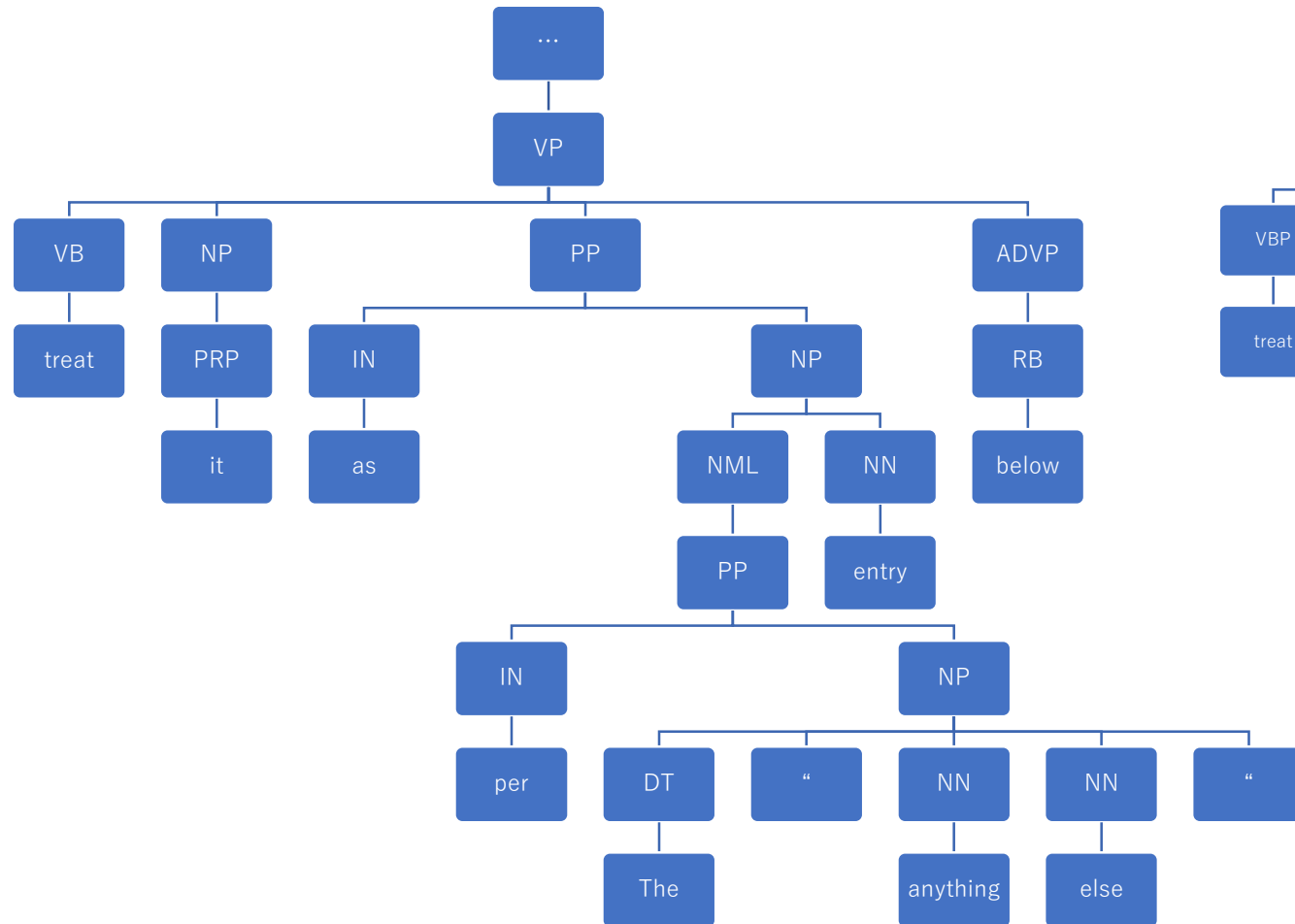


start



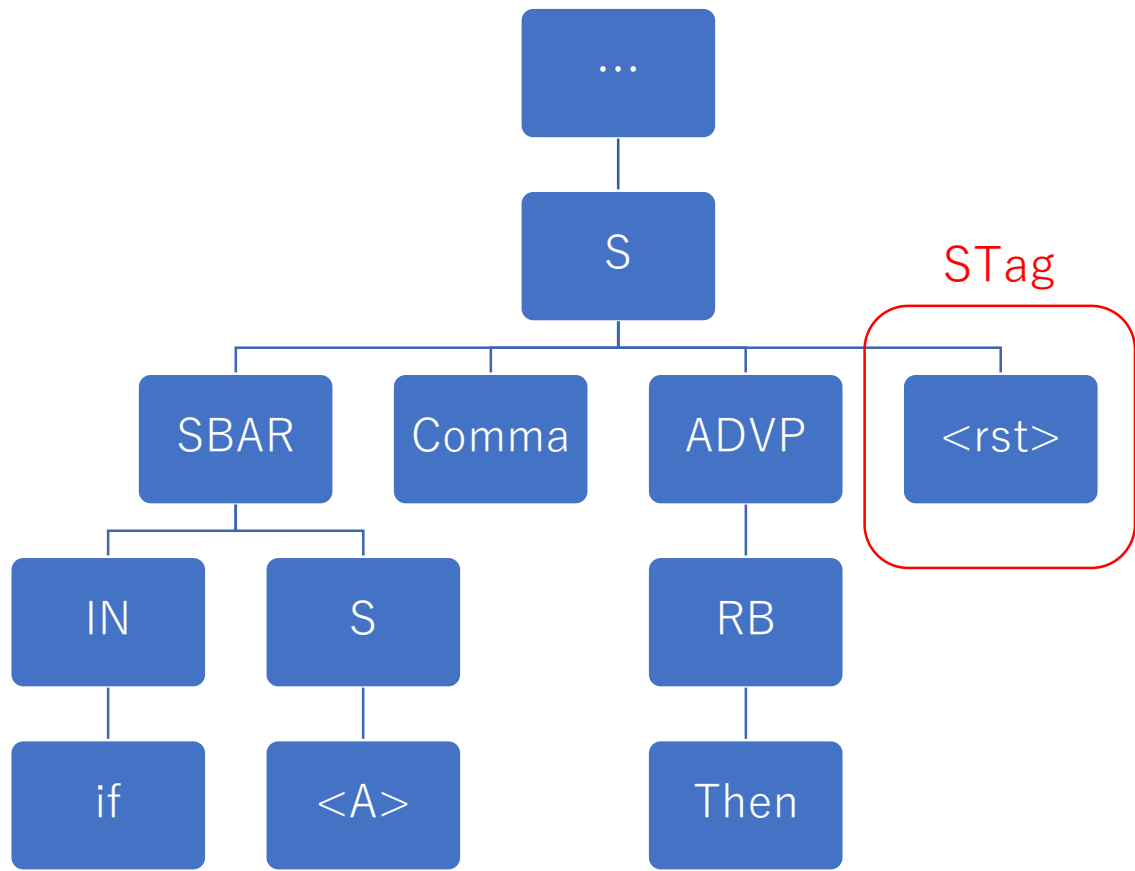
Start(a\_new\_attribute\_in\_the\_current\_tag\_token)

# treat

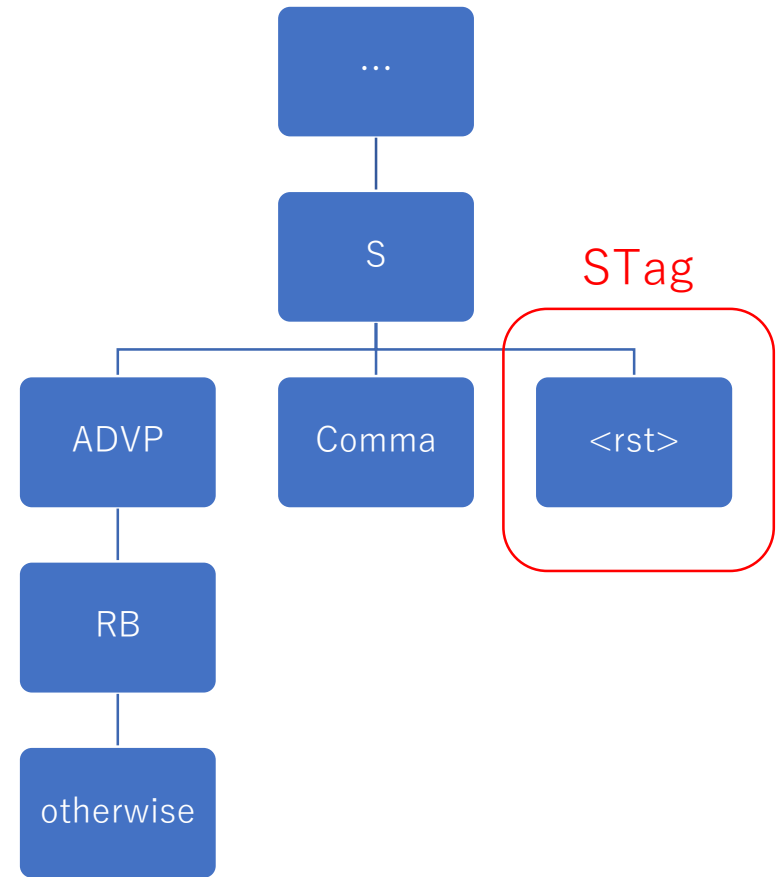


Tree – STag

# If, otherwise

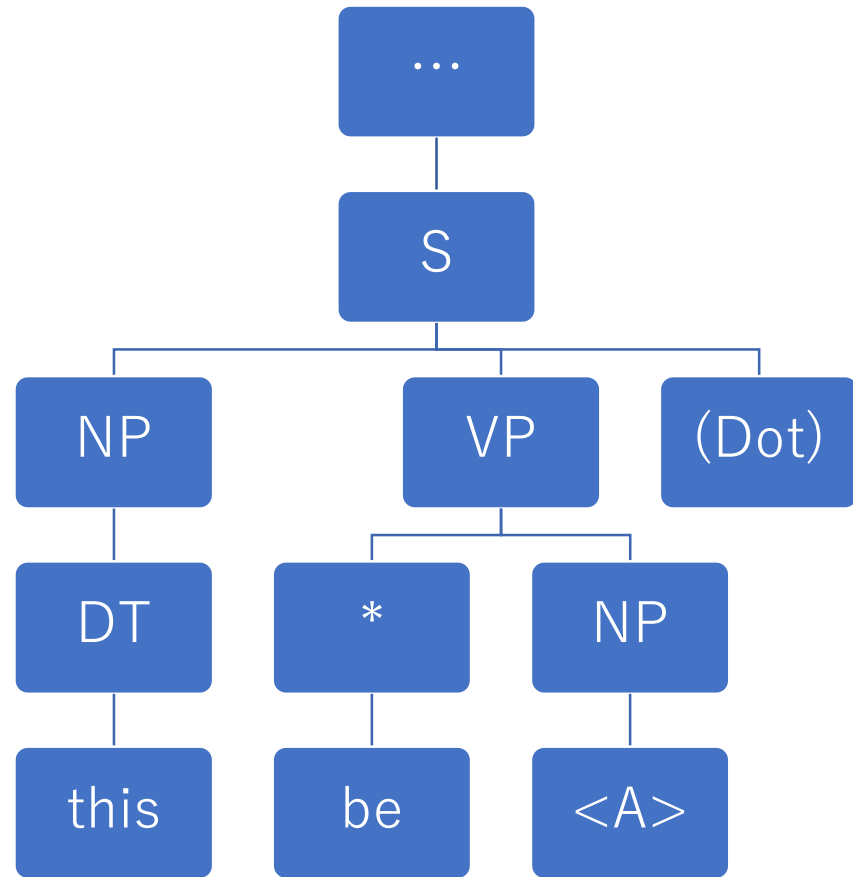


If(<A>, STag(<rst>), null)

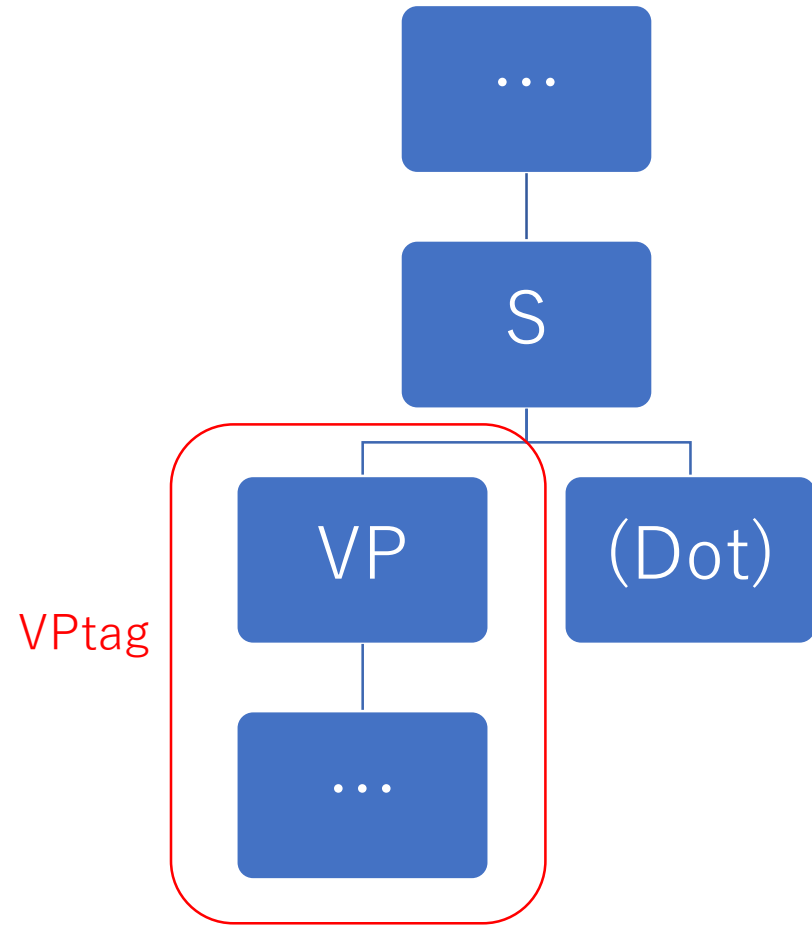


If(null, null, STag(<rst>))

# Error



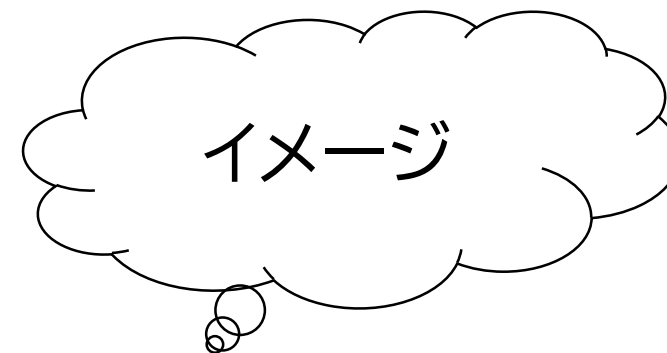
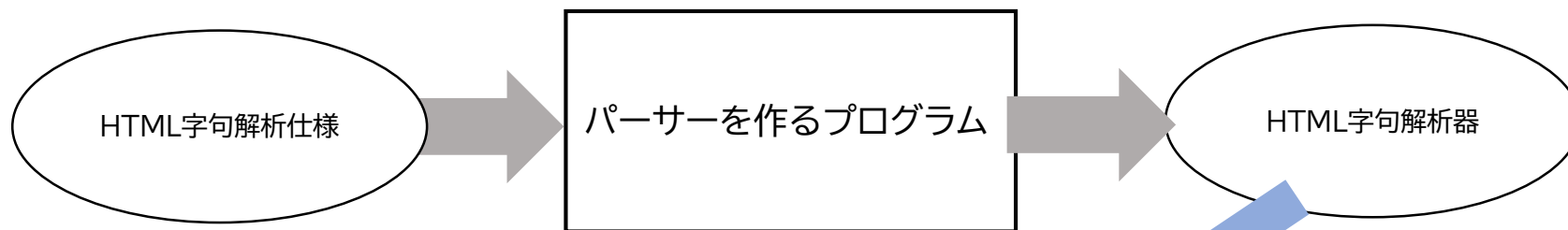
# 命令文



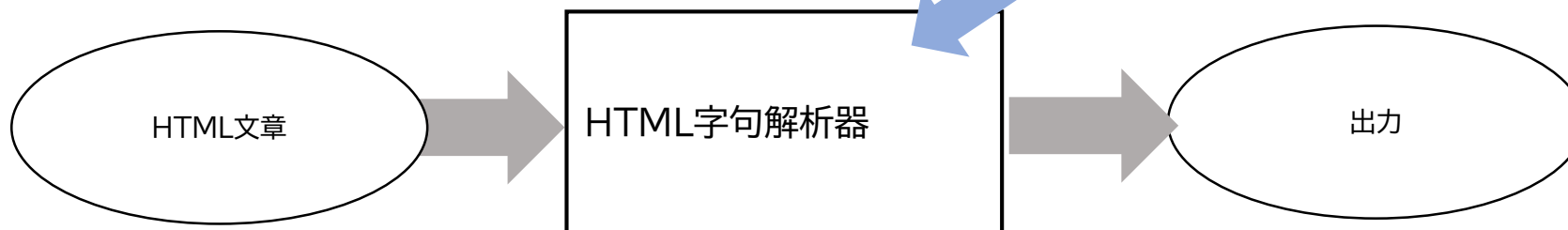
流れ

# 研究のおおまかな感じ

## ① 字句解析器を作る



## ② 字句解析器を実行する(テスト)





# ①の大まかな感じ

## § 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.

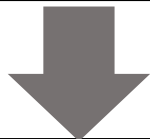
### ↪ **EOF**

Emit an end-of-file token.

### ↪ **Anything else**

Emit the [current input character](#) as a character token.

Scalaの関数



パーサー作成

```
def DataState() = {  
  character match {  
    case "&" => Set(returnState, dataState); Switch(characterReferenceState)  
    case "<" => Switch(tagOpenState)  
    case NULL => Error(unexpected-null-character); Emit(currentInputCharacter)  
    case EOF => Emit(end-of-file)  
    case _ => Emit(currentInputCharacter); Switch(dataState)  
  }  
}
```

# ①-1. HTMLで書かれた字句解析仕様の前処理(構造化する)

HTMLで書かれた字句解析仕様

```
<h5><span>12.2.5.1</span> <dfn>Data state</dfn><a></a></h5>
<p>Consume the <a href=#next-input-character id=data-state:next-input-character>next input
character</a>:</p>
<dl class=switch><dt>U+0026 AMPERSAND (&)<dd>Set the <var id=data-state:return-state><a
href=#return-state>return state</a></var> to the <a href=#data-state id=data-state:data-state>data
state</a>.
Switch to the <a href=#character-reference-state id=data-state:character-reference-state>character
reference state</a>.
```

目的:

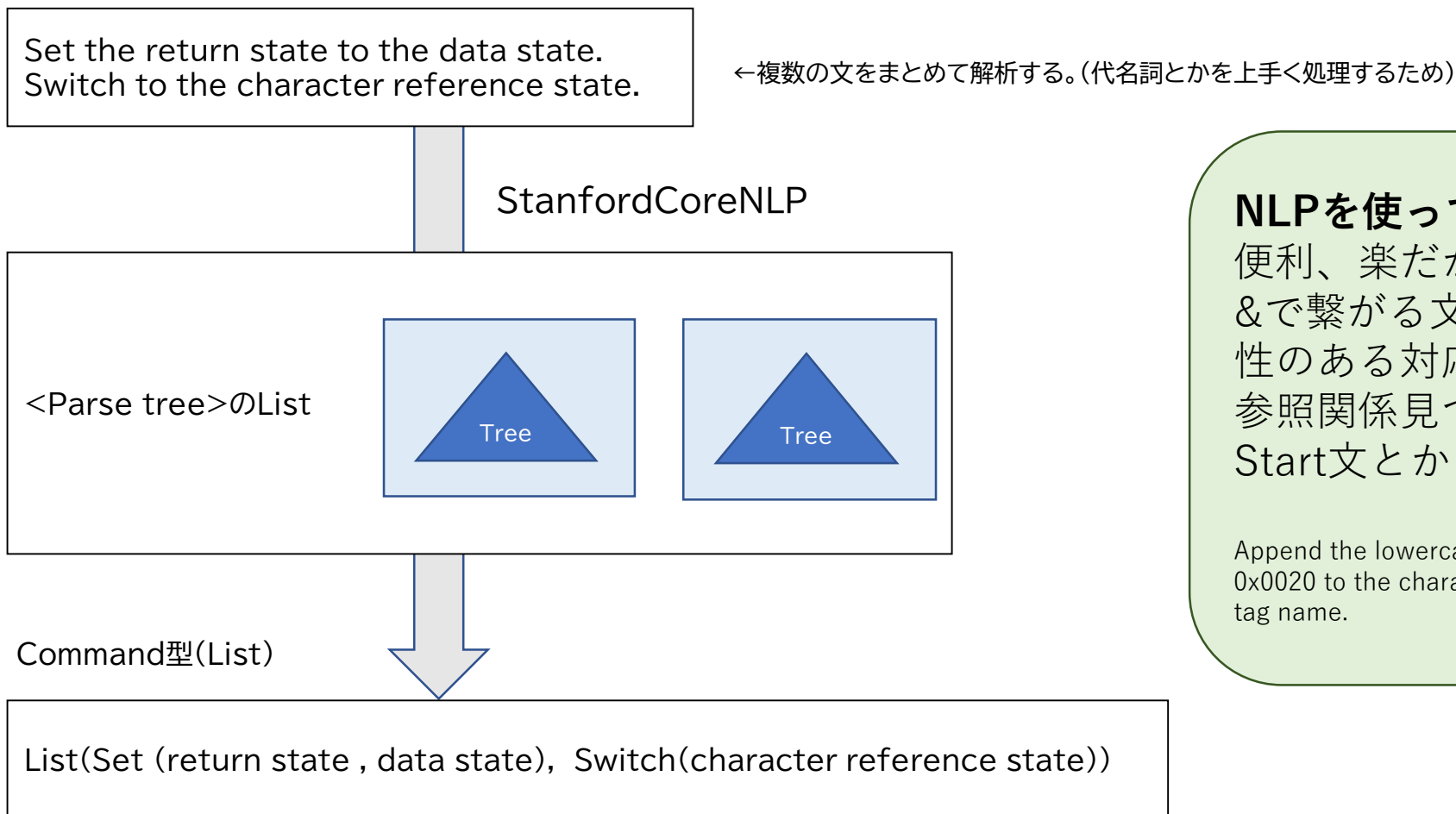
自然言語処理がうまくできるように、前処理として、構造的に書かれている文章をScalaでそのまま改めて構造化する

こんな感じの型に当てはめる

```
class state ( name: String , prev処理: String , 文字を消費する処理: List[trance] )
class trance ( character: string , 処理内容: string )
```

```
state ( Data state , Consume the next input character,
      List ( trance( U+0026, Set the return state to ...), trance(...) )
    )
```

# ① -2. 自然言語で書かれた文章の解析



## NLPを使って解析する理由

便利、楽だから

&で繋がる文、カンマで繋がる文など柔軟性のある対応ができる。

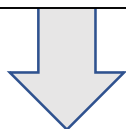
参照関係見つけられる。

Start文とか

Append the lowercase version of the current input character (add 0x0020 to the character's code point) to the current tag token's tag name.

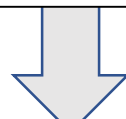
# ①-3. 最終系

```
state ( Data state , Consume the next input character,  
List ( trance( U+0026, Set the return state to ...), trance(...) )  
)
```



自然言語処理

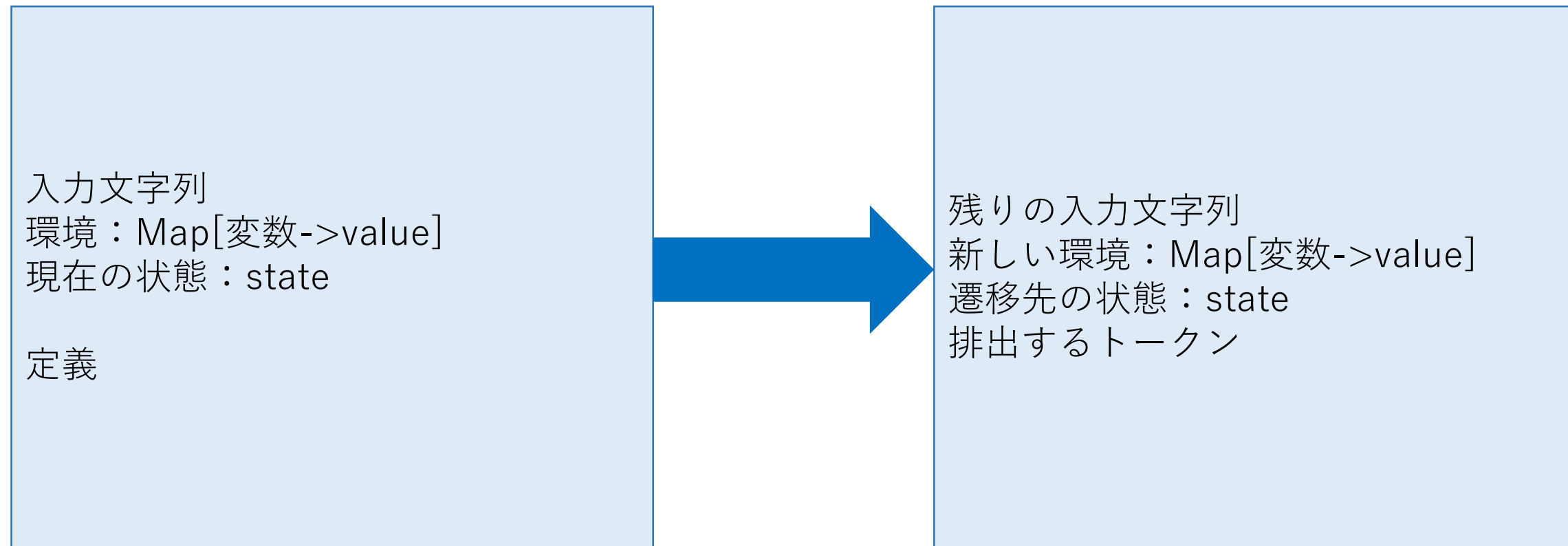
```
state ( Data state , Consume (next input character),  
List ( trance( U+0026, Set (return state , data state) ), trance(...) )  
)
```



何らかの処理

```
def DataState() = {  
  character match {  
    case "&" => Set(returnState, dataState); Switch(characterReferenceState)  
    case "<" => Switch(tagOpenState)  
    case NULL => Error(unexpected-null-character); Emit(currentInputCharacter)  
    case EOF => Emit(end-of-file)  
    case _ => Emit(currentInputCharacter); Switch(dataState)  
  }  
}
```

# 最終系② インタープリタ



77/80個のstateの命令がこの形で書けそう

```
def state名 () = {  
    処理  
    character match {  
        case □ => 処理  
        ...  
    }  
}
```

余談

# 思ったこと

- ・ Html Parseと自然言語処理を同時にやれたら、精度が高くなりそう.

Htmlのタグ構造で単語をまとまり化すれば、変な場所で単語同士のつながりが断たれたりしないと思う.