출처가 명시되지 않은 모든 자료(이미지 등)는 조성현 강사님의 강의 자료 바탕

문장 구조 분석

- `variable` : NP, VP 등 단어 품사, 형태소를 일컫는 용어.
- `terminal` : 실제 단어.
- `derivation` : 왼쪽에 variable, 오른쪽에 terminal이 나오도록 규칙을 정하고, parser를 이용해 parse함.

1. 중의성

`ambiguity`는 중의성으로, NLP에서 문장 구조 분석을 어렵게 하는 요소이다. 다음과 같이 문장(S)을 명사구(NP)와 동사구(VP)로 `derivation`되도록 규칙을 정의하면, 두 가지의 분석 결과가 나온다.

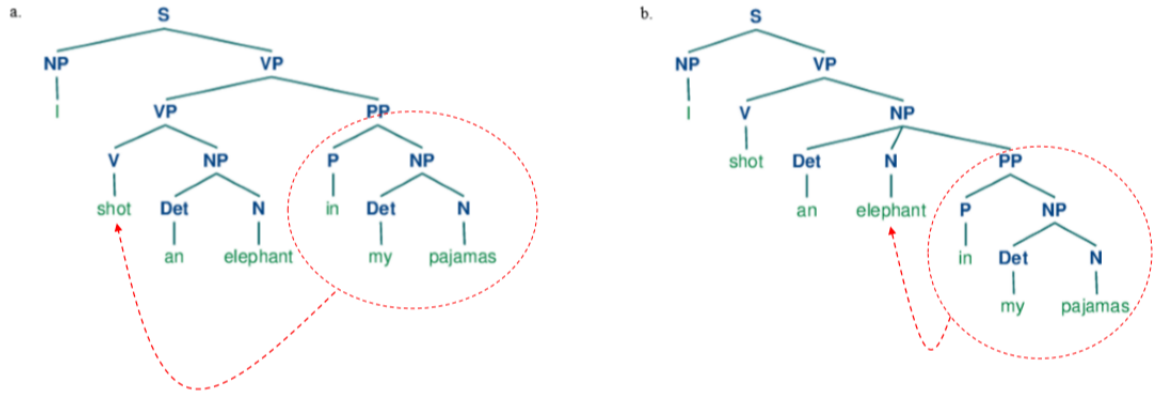
- `CFG` : 문장 string을 읽어 들임.

```
import nltk

grammar = nltk.CFG.fromstring("""
        S -> NP VP
        PP -> P NP
        NP -> Det N | Det N PP | 'I'
        VP -> V NP | VP PP
        Det -> 'an' | 'my'
        N -> 'elephant' | 'pajamas'
        V -> 'shot'
        p -> 'in'
        """)

sentence = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']
parser = nltk.ChartParser(grammar)

for tree in parser.parse(sentence):
    print(tree)
```



- 전치사구(PP)가 동사구(VP)에 걸치게 분석될 수도 있고,
- 전치사구(PP)가 명사(N)에 걸치게 분석될 수도 있다.

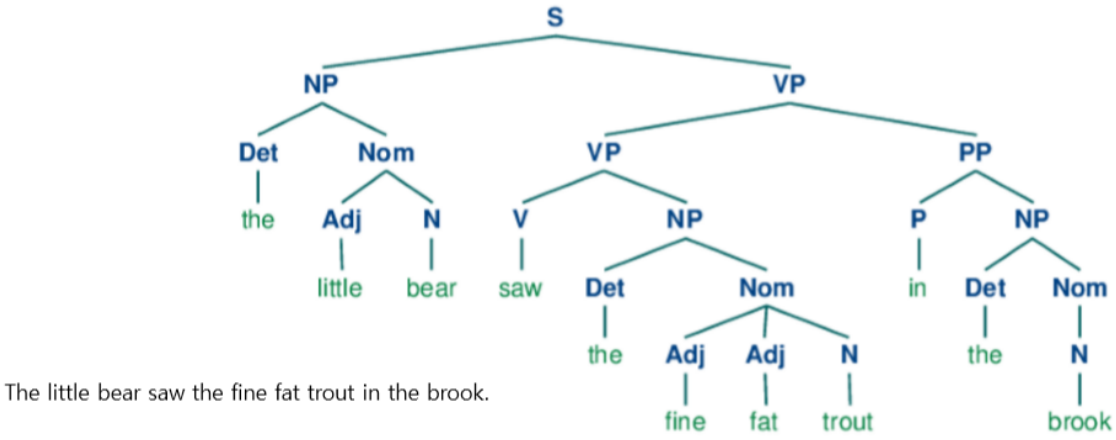
2. 문장 구조

- `constituent` : 통사론에서의 `구성소`. 여러 단어들이 모여서 한 단위로 사용될 수 있는 요소.

아래와 같은 문장에서 `[]`로 묶인 부분이 문장의 `constituent`가 된다.

> 문장: The dog saw the man with a telescope.
>
> - The man saw [the dog with a telescope].
> - The man saw [the dog] [with a telescope].

문장의 구성 성분을 트리 구조로 나타냈을 때, 트리의 각 노드는 `constituent`가 된다.



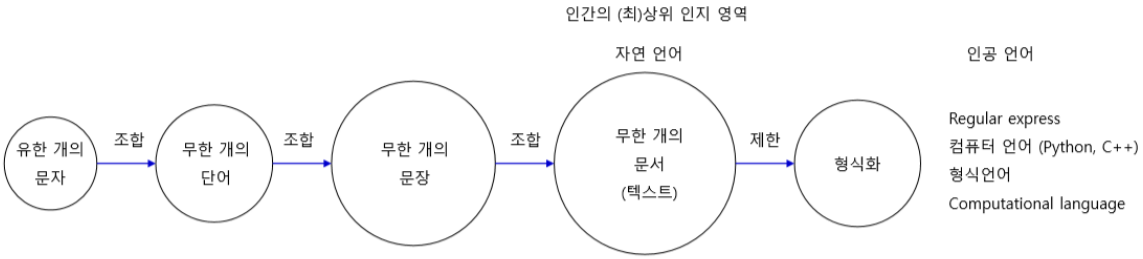
The little bear saw the fine fat trout in the brook.

- `substitution` : 대명사. 묶인 성분이 `constituent`가 되는지 알아보기 위해 `substituion`으로 바꿔 볼 수 있다.

3. 형식언어 이론

언어는 유한 개의 철자로 무한 개의 단어와 문장을 조합한 것으로, 무한한 의미를 생성할 수 있다.

아래 그림에서와 같이 유한 개의 문자를 조합하면 무한 개의 단어가 나오고, 무한 개의 단어를 조합하면 무한 개의 문장이 나오고, 무한 개의 문장을 조합하면 무한 개의 텍스트가 나온다. 인지능력을 가진 인간만이 할 수 있다.



따라서 이러한 인간의 언어를 특정한 *형식*에 따라 *유한하게* 서술할 수 있도록 형식화한다. 이를 **형식언어**(Formal Language)라고 한다. NLP는 인간의 언어를 제한된 형식으로서의 언어인 `형식 언어`로 바꾸어 처리한다. 즉, 인공지능이 인간의 언어를 다 이해할 수 없기 때문에, *제한*을 가해 언어를 *형식화*하여 컴퓨터에서 이해할 수 있도록 하는 것이다.

촘스키의 형식언어 이론

형식언어의 발전에 촘스키의 **형식언어 이론**이 큰 역할을 했다. 특히 이후 인공언어 처리와 연관지어 발전해 나갔다.

촘스키는 문장을 그 내용에 관계 없이, 문장의 형태가 구성되는 과정*만으로*, **형식**만으로서 언어의 계층을 다음의 4가지로 구분했다.

Chomsky Hierarchy

Type	Languages (Grammar)	Production	제한
0	Unrestricted	$\alpha \rightarrow \beta$	$\alpha \in (V+T)^+$ $\beta \in (V+T)^*$
1	Context-sensitive	$\alpha \rightarrow \beta$ $(aAb \rightarrow aYb)$	$\lvert\alpha\rvert \leq \lvert\beta\rvert$ $\alpha, \beta \in (V+T)^+$ $a, b : context$
2	Context-free	$A \rightarrow \alpha$	$A \in V$ $\alpha \in (V+T)^*$
3	Regular	$A \rightarrow a$ $A \rightarrow aB$	$A, B \in V$ $a \in \Sigma^*$

제한 없음
개수 제한
좌변 제한
우변까지 제한

$V : Variable, T : Terminal, \Sigma : finite\ alphabet$

제한이 없는 인간의 언어와 같은 언어의 타입을 `0`(Unrestricted)으로, 컴퓨터 언어에서의 정규표현식과 같이 가장 제약이 많은 언어의 타입을 `3`(Regular)으로 구분했다. 그 사이에서 제약이 어느 정도로 많아지는지 그 정도에 따라 `1`(Context-Sensitive), `2`(Context-Free)로 구분했다. `3`에서 `1`로 올 수록 high level 언어이다.

- `0` : Unrestricted

아무런 제한 없이 `derivation`되는 언어. 무제한 언어. 좌변에서 우변으로 가는 데 아무 것이나 올 수 있고(asterisk(`*`)로 표현), 그 개수에도 제한이 없이 어떻게나 반복될 수 있다.

- `1` : Context-Sensitive

좌변의 개수가 우변의 개수보다 작거나 같아야 한다는 개수 제한을 둔다. 다만 문장 주변에 어떠한 context가 오는지에 따라 `derivation`이 달라진다. 예컨대, `aSb -> NP VP`와 같은 식으로 문장(S) 앞에 a가 오고 뒤에 b가 온다면, 우변과 같이 `derivation`된다고 정하는 방식이다. 똑같은 S가 `derivation`되더라도 `cSd -> NP PP`와 같은 규칙에 의해 `derivation`과 같이 주변 문맥에 의해 다른 형태일 수도 있다. 아직까지 문맥에 영향을 받으므로 인간의 언어에 가깝다.

- `2` : Context-Free

좌변의 개수를 1개로 제한한다. 문맥의 영향을 없앤 것이다. 중의성을 살펴 보기 위해 진행한 실습에서 `S -> NP VP`와 같이 정의한 것이 예이다.

- `3` : Regular

우변의 개수까지 제한한다. 완벽하게 형식에 맞아야 하며, 가장 인공언어에 가까운 저수준 언어이다.

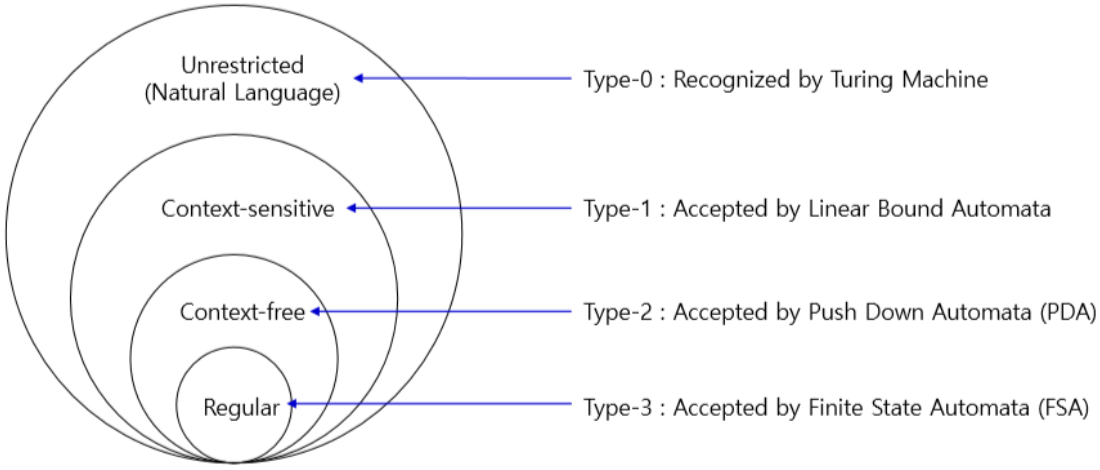
오토마타 구성

- `오토마타` : 컴퓨터 과학에서 입력과 내부 출력 간의 관계를 수학 모델로 옮기고, 이를 신호 또는 동작의 형태로 외부에 출력하는 것.

> 그냥 `오토마타`가 그런 거라고 알아만 둡시다.

언어가 어떤 타입에 속하는지 판별하는 것을 *Membership Problem*이라고 하는데, 이를 판단하기 위한 장치로서 오토마타를 사용한다.

촘스키의 계층 구조 이론에 따라 각 타입의 언어를 이해할 수 있는 오토마타의 구성도 다음과 같이 달라진다.



수업에서는 `3` 유형의 유한 상태 오토마타인 `FSA`만을 간단히 구현해 보도록 한다.

FSA(Finite State Acceptor)

형식이 엄격하게 제한된 정규언어를 이해하는 기계이다. `Acceptor`에서도 알 수 있듯, 제한된 정규언어의 표현을 따르면 `accept`, 따르지 않으면 `reject`한다.

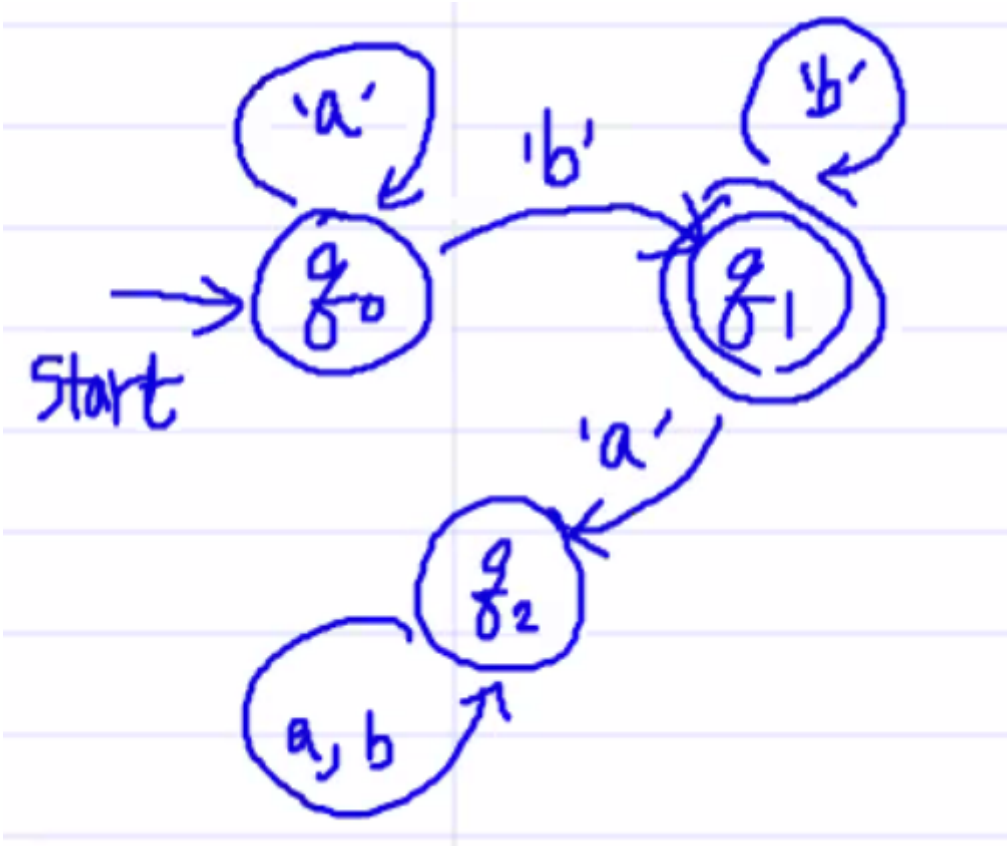
다음과 같은 요소로 구성되어 있다.

- `Σ`: 나올 수 있는 모든 문자열의 집합.
- `Q` : 가능한 상태.
  - `q0` : 초기 상태. 원으로 나타냄.
  - `q1`, `q2`, ... `qn-1` : 각각의 상태. 원으로 나타냄.
  - `qn` : 마지막 상태. 그림에서는 이중 원으로 나타냄.
- `ϵ` : 아무 것도 등장하지 않는 상태.

- δ : transition. 각각의 상태에서 어떤 규칙을 따를 때 다른 상태로 이동하는지 규칙을 나타냄. 화살표로 나타냄.

작동 원리는 간단하다. 정의된 transition 규칙에 따라 상태가 이동한다. 해당 규칙에 맞지 않는 상태가 나오면 `trap state`에 갇혀 종료된다. 문자열, 문장의 구성 요소들을 읽어 들이다가 `Accept` 규칙에 맞지 않으면 바로 위배되는 것이라 이해해 종료하는 것이다.

`a`, `b` 2개의 문자열이 나오고 `b`로 끝나도록 하는 오토마타를 구현해 보자.




```
init_state = 0
final_state = [1]
trap_state = 2

delta = {0: {'a': 0, 'b': 1},
         1: {'a': 2, 'b': 1}}

def FSA(string):
    state = init_state
    for s in string:
        state = delta[state][s]
        if state == trap_state:
            break
    return state in final_state
```

위와 같은 간단한 FSA 오토마타를 구현했을 때, 만약 `state`가 `final_state`에서 끝났으면 `True`를, 아니면 `False`를 반환한다.

> *참고*
>
> 위와 같은 오토마타를 정규표현식과 `Derivation` grammar로 표현하면 다음과 같다.
>
> - 정규표현식 : `a*b+`
> - 문법 : `S -> aS|aA`, `A -> aA|aB`

Transducer

유한 상태 변환기로서, 입력 값의 상태가 변화할 때 특정 값을 출력하는 것을 의미한다.

참고

언어 이론 부분은 여기서 마친다. 뒷부분은 생략한다. 한글 오토마타에 대해 더 알아 보고 싶다면 여기를, 한글 유한 상태 변환기에 대해 더 알아 보고 싶다면 이 강의를 참고하자.

Rouzeta : 이상호 박사님 강연

1. 형태소 분석기
    - **형태소 분석기**: 모든 가능한 형태소 분석 결과를 내 주는 모듈.
    - **품사 태깅**: 형태소 분석기를 통해 얻을 수 있는 문장 내 모든 형태소 분석 결과 중 *가장 적절한* 것을 선택.

2. 형태소 분석기 활용 분야: NLP 처리의 기본 중 기본
    - 검색 시스템 색인 추출
    - 음성 인식 및 합성

3. KTS 오픈소스 형태소 분석기 vs. Transducer
    - KTS: 자료구조 만들고, 메모리에 올려 다익스트라 알고리즘으로 shortest path 찾아 품사 태깅.
    - Transducer: 시간복잡도 O(N)일 뿐만 아니라, 사람이 생각하는 것과 굉장히 유사한 방식에 충격.

> *" 인간이 문장을 읽을 때 ambiguity 를 만나면 그 때 결정하지 않고 ambiguity 를 쌓아 놓았다가 어느 순간 확실한 단서 (clue) 를 만나면 그동안 해결하지 못한 것을 한꺼번에 해결하는 것과 같은 작동 방식."*