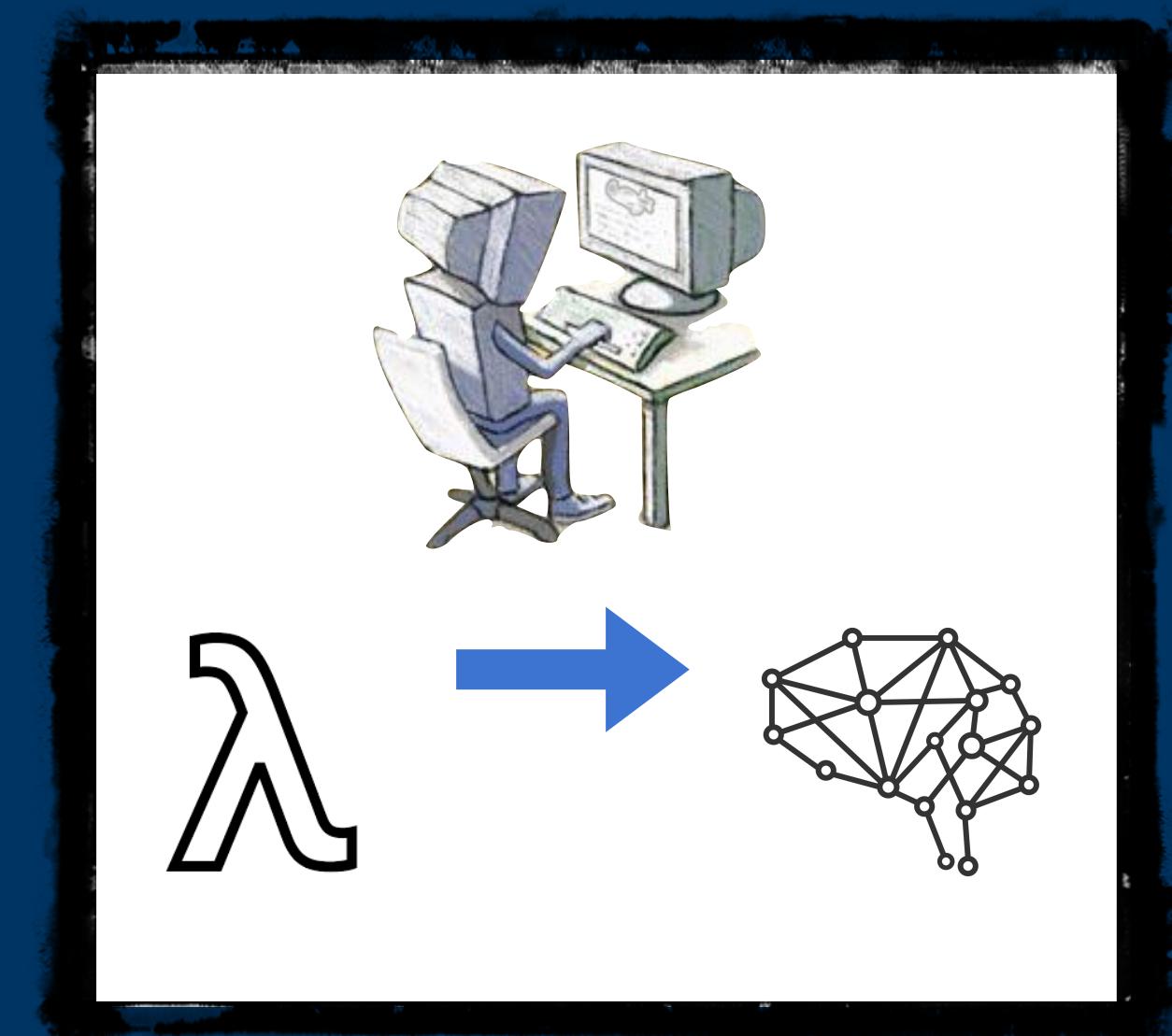


# 속성이 주도하는 제약 생성으로 의미 지키기: 변수명 오용 방지와 함께

박준성, 김진상, 이우석  
한양대학교 프로그래밍시스템 연구실 (PSL)



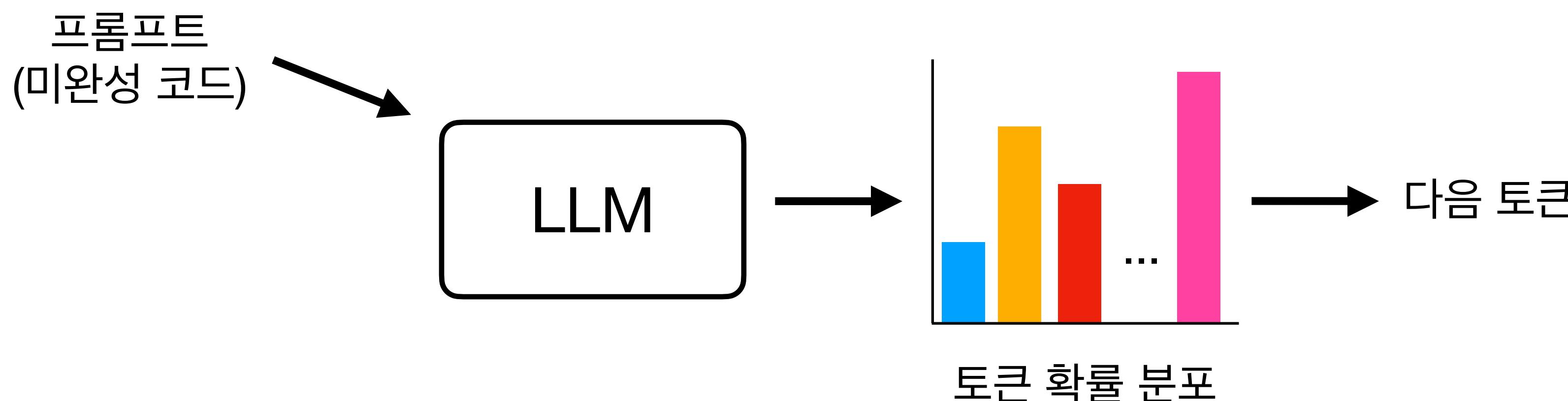
# 배경 지식

제약 생성(Constrained decoding)이란?

# 배경 지식

## 제약 생성(Constrained decoding)이란?

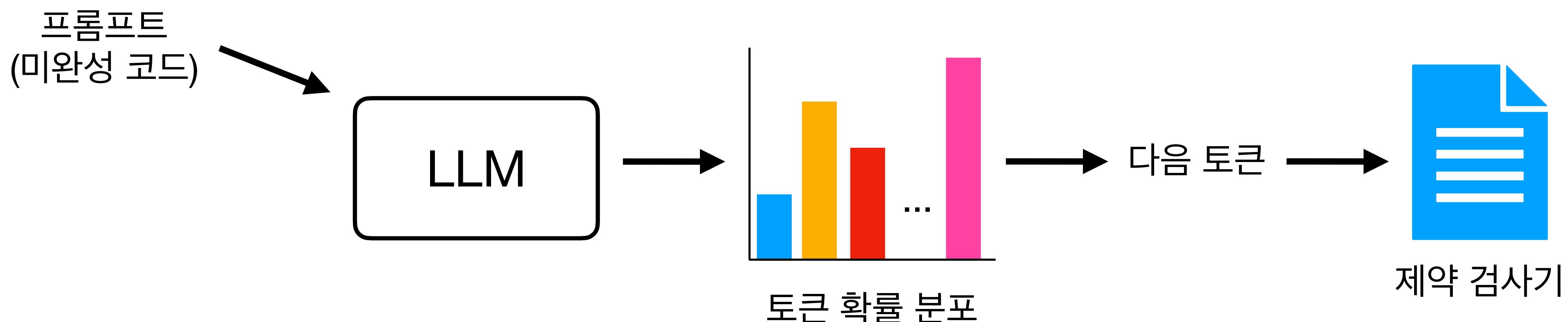
- LLM의 코드 생성 과정 중  
특정 규칙이나 제약을 만족하도록 다음 토큰 선택을 제한하는 방법



# 배경 지식

## 제약 생성(Constrained decoding)이란?

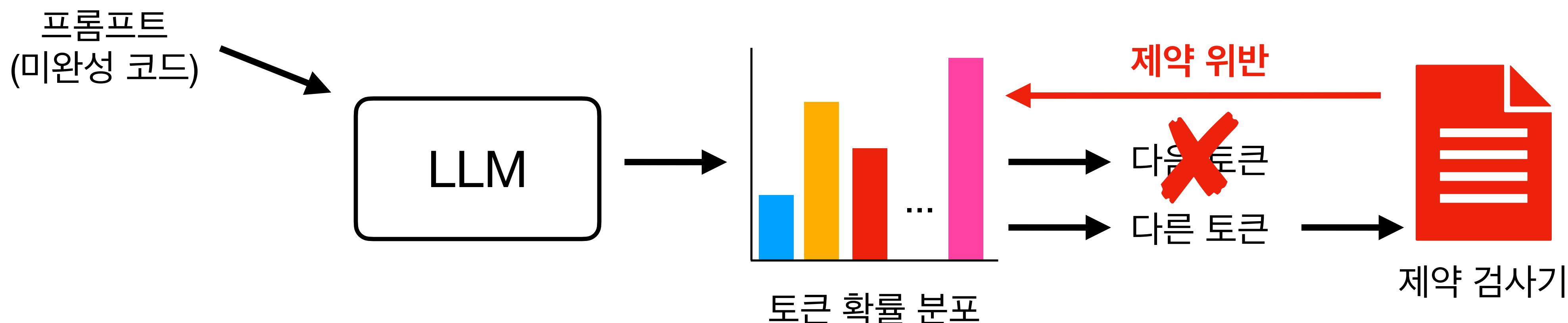
- LLM의 코드 생성 과정 중  
특정 규칙이나 제약을 만족하도록 다음 토큰 선택을 제한하는 방법



# 배경 지식

## 제약 생성(Constrained decoding)이란?

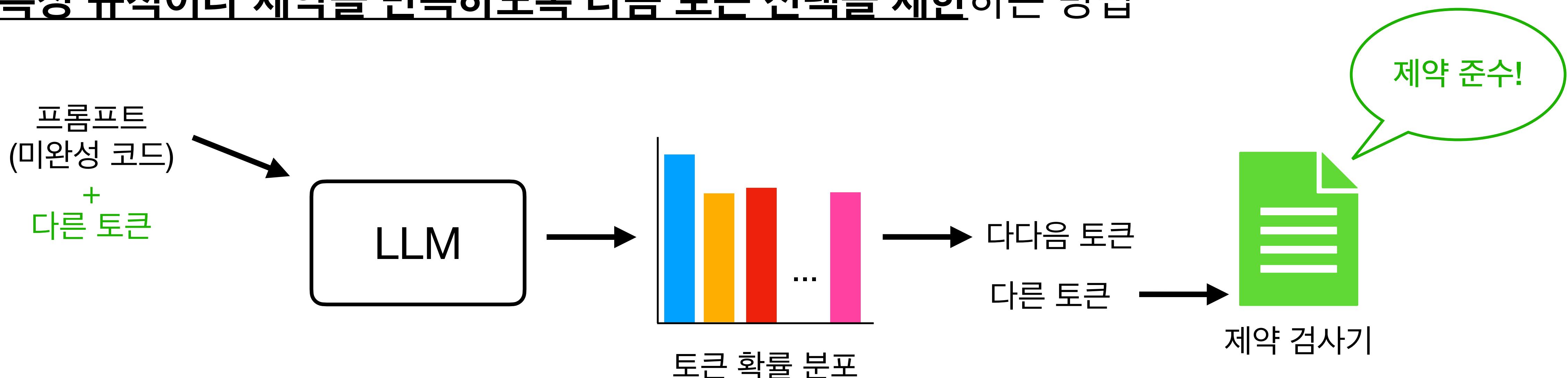
- LLM의 코드 생성 과정 중  
특정 규칙이나 제약을 만족하도록 다음 토큰 선택을 제한하는 방법



# 배경 지식

## 제약 생성(Constrained decoding)이란?

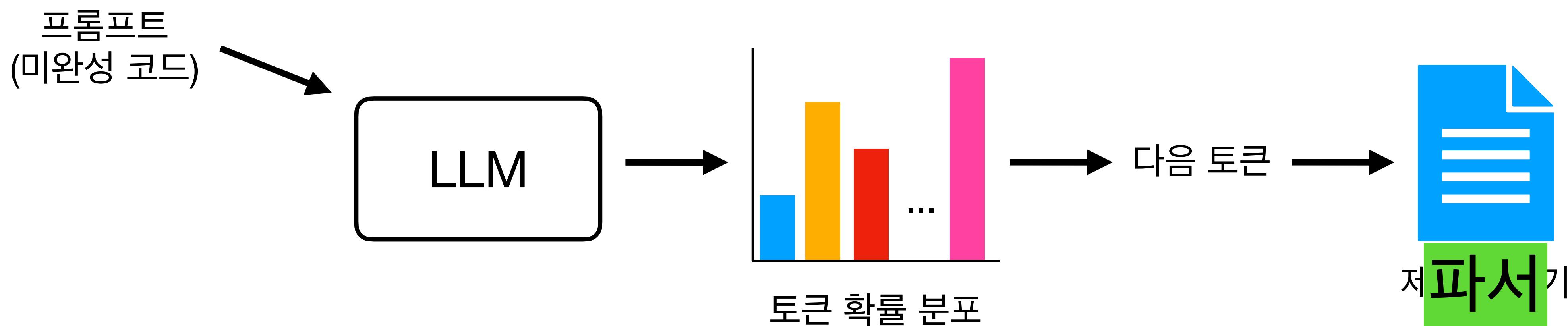
- LLM의 코드 생성 과정 중  
특정 규칙이나 제약을 만족하도록 다음 토큰 선택을 제한하는 방법



# 배경 지식

## 제약 생성(Constrained decoding)이란?

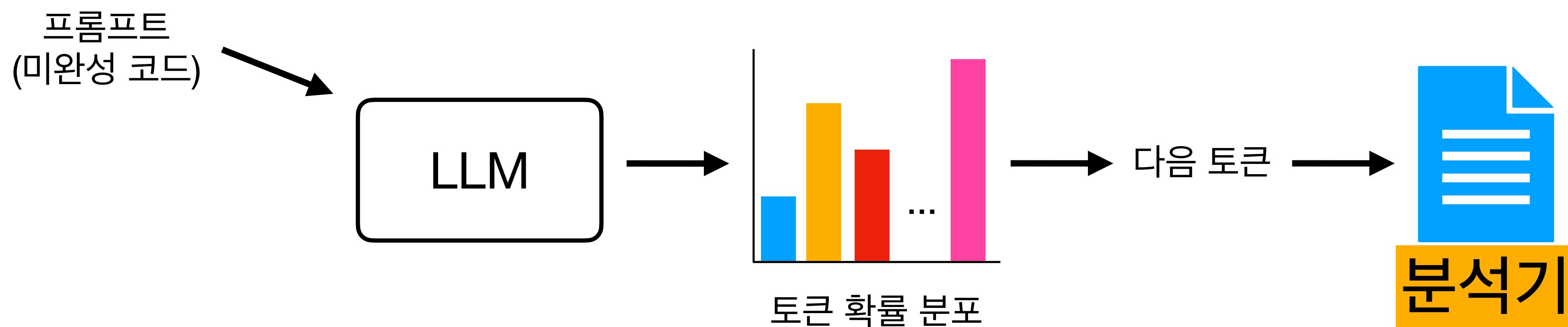
- LLM의 코드 생성 과정 중  
**문법적 올바름**을 만족하도록 다음 토큰 선택을 제한하는 방법



# 배경 지식

## 제약 생성(Constrained decoding)이란?

- LLM의 코드 생성 과정 중  
**의미적 올바름**을 만족하도록 다음 토큰 선택을 제한하는 방법



# 배경 지식

## 기존 방법의 한계

- 대부분의 제약 생성 방법론이 문법적 올바름만 보장

# 배경 지식

## 기존 방법의 한계

- 대부분의 제약 생성 방법론이 문법적 올바름만 보장
- 그 외 의미적 올바름을 보장하는 기존 제약 생성 방법은
  1. 매 토큰 생성 단계마다 지금까지 생성한 코드 전부를 분석
  2. 목표 언어와 특성을 위한 분석기를 직접 만듦

# 배경 지식

## 기존 방법의 한계

- 대부분의 제약 생성 방법론이 문법적 올바름만 보장
- 그 외 의미적 올바름을 보장하는 기존 제약 생성 방법은
  1. 매 토큰 생성 단계마다 지금까지 생성한 코드 전부를 분석  
분석이 언제, 어디에 필요한지는 속성을 통해 판단하고,
  2. 목표 언어와 특성을 위한 분석기를 직접 만듦  
기존의 분석기를 활용하여 의미적 올바름 보장

속성이 주도하는 제약 생성!

# 여기서 드는 의문점

여기서 드는 의문점

속성이 뭔데

여기서 드는 의문점

속성이 뭔데

그게 꼭 필요해?

여기서 드는 의문점

속성이 뭔데

그게 꼭 필요해? ←

**꼭 '속성'이 필요한가?**

**그냥 안전한 분석기로 대충 하면 안돼?**

# 꼭 '속성'이 필요한가?

그냥 안전한 분석기로 대충 하면 안돼?

- ex. 변수명 오류: 다음 미완성 Python 코드만 보고 변수명 오사용을 미리 판단해보기
  - [ i

# 꼭 '속성'이 필요한가?

그냥 안전한 분석기로 대충 하면 안돼?

- ex. 변수명 오류: 다음 미완성 Python 코드만 보고 변수명 오사용을 미리 판단해보기
  - [ i

```
1. [ i ] # undefined variable 'i'
```

# 꼭 '속성'이 필요한가? 그냥 안전한 분석기로 대충 하면 안돼?

- ex. 변수명 오류: 다음 미완성 Python 코드만 보고 변수명 오사용을 미리 판단해보기
  - [ i
    - 1. [ i ] # undefined variable 'i'
    - 2. [i for i in range(10) ] # no variable error
  - i의 참조가 먼저 일어나고, 나중에 선언되어 변수명 오사용이 해결되는 케이스

# 꼭 '속성'이 필요한가?

그냥 안전한 분석기로 대충 하면 안돼?

- ex. 변수명 오류: 다음 미완성 Python 코드만 보고 변수명 오사용을 미리 판단해보기

- a

1. a () # undefined variable 'a'

2. a = 0 # no variable misuse

# 꼭 '속성'이 필요한가?

그냥 안전한 분석기로 대충 하면 안돼?

- ex. 변수명 오류: 다음 미완성 Python 코드만 보고 변수명 오사용을 미리 판단해보기

- a

```
1. a() # undefined variable 'a'
```

```
2. a = 0 # no variable misuse
```

- 1번 케이스 때문에 안전한 분석기는 a를 막도록 결정

# 꼭 '속성'이 필요한가?

그냥 안전한 분석기로 대충 하면 안돼?

- ex. 변수명 오류: 다음 미완성 Python 코드만 보고 변수명 오사용을 미리 판단해보기

- a

```
1. a() # undefined variable 'a'
```

```
2. a = 0 # no variable misuse
```

- 1번 케이스 때문에 안전한 분석기는 a를 막도록 결정

- 분석기를 매번 사용하는 것만이 능사는 아니다. 추가적인 정보가 필요.

# 그럼 '속성'이 뭔데?

정의

- 속성달린 문법(attribute grammar)에서

생김새 규칙과 더불어 의미 규칙을 표현하기 위한

문법 기호에 값을 저장할 수 있는 변수

# 그럼 '속성'이 뭔데?

## 예제로 알아보는 속성

- 속성달린 문법(attribute grammar)에서

생김새 규칙과 더불어 의미 규칙을 표현하기 위한

문법 기호에 값을 저장할 수 있는 변수

```
start := stmt
stmt   := LPAR stmt1
        | RPAR stmt1
        | EMPTY
```

괄호 언어의 생김새 규칙

# 그럼 '속성'이 뭔데?

## 예제로 알아보는 속성

- 속성달린 문법(attribute grammar)에서

생김새 규칙과 더불어 의미 규칙을 표현하기 위한

문법 기호에 값을 저장할 수 있는 변수

```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

괄호 언어의 생김새 규칙

```
stmt.in  := 0           , start.out := stmt.out
stmt1.in := stmt.in + 1, stmt.out   := stmt1.out
stmt1.in := stmt.in - 1, stmt.out   := stmt1.out
                           stmt.out := stmt.in
```

괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데?

## 예제로 알아보는 속성

- 속성달린 문법(attribute grammar)에서

생김새 규칙과 더불어 의미 규칙을 표현하기 위한

문법 기호에 값을 저장할 수 있는 변수

in: 지금까지 닫히지 않은 괄호의 수

```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

괄호 언어의 생김새 규칙

```
stmt.in  := 0 , start.out := stmt.out
stmt1.in := stmt.in + 1, stmt.out  := stmt1.out
stmt1.in := stmt.in - 1, stmt.out  := stmt1.out
                           stmt.out := stmt.in
```

괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데?

## 예제로 알아보는 속성

- 속성달린 문법(attribute grammar)에서

생김새 규칙과 더불어 의미 규칙을 표현하기 위한

문법 기호에 값을 저장할 수 있는 변수

```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

괄호 언어의 생김새 규칙

in: 지금까지 닫히지 않은 괄호의 수

```
stmt.in := 0
stmt1.in := stmt.in + 1,
stmt1.in := stmt.in - 1,
```

out: 아직도 닫히지 않은 괄호의 수

```
start.out := stmt.out
stmt.out := stmt1.out
stmt.out := stmt1.out
stmt.out := stmt.in
```

괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데?

파싱으로 정해지는 속성의 값

↓  
( ) ) \$  
문자열

< start : • stmt | Ø >  
**파서 스택**

in: 지금까지 닫히지 않은 괄호의 수

out: 아직도 닫히지 않은 괄호의 수

```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

```
stmt.in  := 0 , start.out := stmt.out
stmt1.in := stmt.in + 1, stmt.out  := stmt1.out
stmt1.in := stmt.in - 1, stmt.out  := stmt1.out
                                         stmt.out := stmt.in
```

괄호 언어의 **생김새 규칙**

괄호 언어의 **의미 규칙**

# 그럼 '속성'이 뭈데?

파싱으로 정해지는 속성의 값

↓  
( ) ) \$  
문자열

< stmt : • LPAR stmt<sub>1</sub> | stmt.in ↣ 0 >  
< start : • stmt | Ø >  
파서 스택

in: 지금까지 닫히지 않은 괄호의 수

out: 아직도 닫히지 않은 괄호의 수

start := stmt  
stmt := LPAR stmt<sub>1</sub>  
| RPAR stmt<sub>1</sub>  
| EMPTY

stmt.in := 0 , start.out := stmt.out  
stmt<sub>1</sub>.in := stmt.in + 1, stmt.out := stmt<sub>1</sub>.out  
stmt<sub>1</sub>.in := stmt.in - 1, stmt.out := stmt<sub>1</sub>.out  
stmt.out := stmt.in

괄호 언어의 생김새 규칙

괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데?

파싱으로 정해지는 속성의 값

↓  
( ) ) \$  
문자열

< stmt : LPAR • stmt<sub>1</sub> | stmt.in ↣ 0 >  
< start : • stmt | Ø >  
**파서 스택**

in: 지금까지 닫히지 않은 괄호의 수

out: 아직도 닫히지 않은 괄호의 수

```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

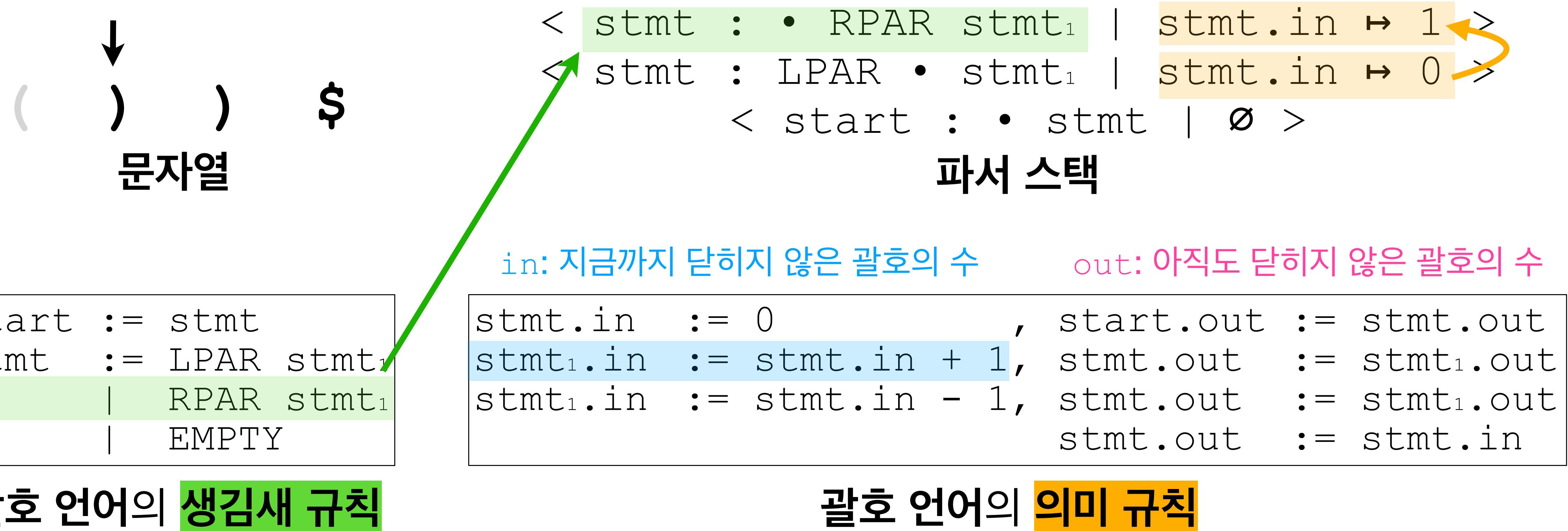
```
stmt.in  := 0 , start.out := stmt.out
stmt1.in := stmt.in + 1, stmt.out  := stmt1.out
stmt1.in := stmt.in - 1, stmt.out  := stmt1.out
                                         stmt.out := stmt.in
```

괄호 언어의 생김새 규칙

괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데?

파싱으로 정해지는 속성의 값



# 그럼 '속성'이 뭔데?

파싱으로 정해지는 속성의 값

( ) ) \$

문자열

< stmt : • EMPTY | stmt.in  $\mapsto$  -1 >  
< stmt : RPAR • stmt<sub>1</sub> | stmt.in  $\mapsto$  0 >  
< stmt : RPAR • stmt<sub>1</sub> | stmt.in  $\mapsto$  1 >  
< stmt : LPAR • stmt<sub>1</sub> | stmt.in  $\mapsto$  0 >  
< start := • stmt | Ø >

파서 스택

in: 지금까지 닫히지 않은 괄호의 수

out: 아직도 닫히지 않은 괄호의 수

```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

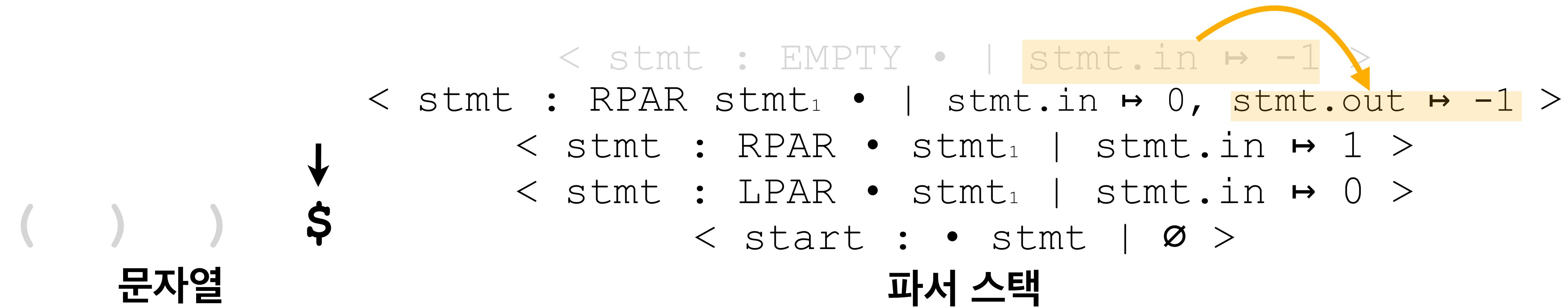
```
stmt.in  := 0 , start.out := stmt.out
stmt1.in := stmt.in + 1, stmt.out := stmt1.out
stmt1.in := stmt.in - 1, stmt.out := stmt1.out
                                         stmt.out := stmt.in
```

괄호 언어의 생김새 규칙

괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데?

파싱으로 정해지는 속성의 값



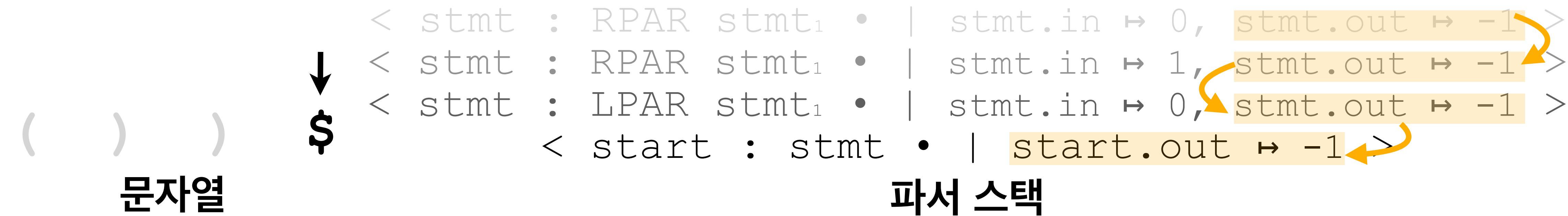
```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

괄호 언어의 생김새 규칙

```
stmt.in  := 0           , start.out  := stmt.out
stmt1.in := stmt.in + 1, stmt.out   := stmt1.out
stmt1.in := stmt.in - 1, stmt.out   := stmt1.out
                                         stmt.out  := stmt.in
```

괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데? 파싱으로 정해지는 속성의 값



```
start ::= stmt
stmt   ::= LPAR stmt
          | RPAR stmt
          | EMPTY
```

# 괄호 언어의 생김새 규칙

```
stmt.in    := 0           , start.out := stmt.out  
stmt1.in   := stmt.in + 1, stmt.out    := stmt1.out  
stmt1.in   := stmt.in - 1, stmt.out    := stmt1.out  
                           stmt.out    := stmt.in
```

# 괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데?

파싱으로 정해지는 속성의 값

- 아직도 (끝까지) 닫히지 않은 괄호의 수: -1 (즉, 괄호 하나를 더 많이 닫음)

( ) ) \$  
문자열

< start := stmt • | start.out ↪ -1 >  
파서 스택

in: 지금까지 닫히지 않은 괄호의 수

out: 아직도 닫히지 않은 괄호의 수

```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

```
stmt.in  := 0 , start.out := stmt.out
stmt1.in := stmt.in + 1, stmt.out  := stmt1.out
stmt1.in := stmt.in - 1, stmt.out  := stmt1.out
                                         start.out := stmt.in
```

괄호 언어의 생김새 규칙

괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데?

## 속성으로 의미적 올바름 검사하기

- 괄호 언어가 의미적으로 올바르다는 건, "모든 괄호 쌍이 균형잡힌 상태"라고 하자.

```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

괄호 언어의 생김새 규칙

in: 지금까지 닫히지 않은 괄호의 수

```
stmt.in := 0 , start.out := stmt.out
stmt1.in := stmt.in + 1, stmt.out := stmt1.out
stmt1.in := stmt.in - 1, stmt.out := stmt1.out
                                         stmt.out := stmt.in
```

괄호 언어의 의미 규칙

# 그럼 '속성'이 뭔데?

## 속성으로 의미적 올바름 검사하기

- 괄호 언어가 의미적으로 올바르다는 건, "모든 괄호 쌍이 균형잡힌 상태"라고 하자.
- 새로운 속성 `ok`: 괄호 언어가 의미적으로 올바르면 `True`, 아니면 `False`

```
start := stmt
stmt  := LPAR stmt1
      | RPAR stmt1
      | EMPTY
```

괄호 언어의 **생김새 규칙**

```
..., start.out := stmt.out , start.ok := stmt.ok
..., stmt.out  := stmt1.out , stmt.ok  := stmt.out ≥ 0
..., stmt.out  := stmt1.out , stmt.ok  := stmt.out ≥ 0
        stmt.out  := stmt.in  , stmt.ok  := stmt.out = 0
```

괄호 언어의 **의미 규칙**

out: 아직도 닫히지 않은 괄호의 수

out: 의미적으로 올바른지 여부

# '속성'은 만능인가?

## 바퀴의 재발명

- 속성과 의미 규칙을 잘 정의하면 의미적 올바름을 보장할 수도 있음

# '속성'은 만능인가?

## 바퀴의 재발명

- 속성과 의미 규칙을 잘 정의하면 의미적 올바름을 보장할 수도 있음
- 하지만, 복잡한 언어에 대해서는 잘 정의하는 것이 어렵고,

# '속성'은 만능인가?

## 바퀴의 재발명

- 속성과 의미 규칙을 잘 정의하면 의미적 올바름을 보장할 수도 있음
- 하지만, 복잡한 언어에 대해서는 잘 정의하는 것이 어렵고,  
이미 존재하는 기존의 정적 분석기를 다시 만드는 것과 다르지 않음

# '속성'은 만능인가?

## 바퀴의 재발명

- 속성과 의미 규칙을 잘 정의하면 의미적 올바름을 보장할 수도 있음
- 하지만, 복잡한 언어에 대해서는 잘 정의하는 것이 어렵고, 이미 존재하는 기존의 정적 분석기를 다시 만드는 것과 다르지 않음
- 따라서, 속성은 언제, 어디서 분석을 진행할지를 알려주는 지표 역할로 사용하고 속성에 의해 분석이 필요한 상황임이 밝혀지면, 이미 존재하는 분석기를 활용!

# 속성이 주도하는 제약 생성

## 다시, 괄호 언어

```
start ::= stmt
stmt  ::= LPAR stmt1
        | RPAR stmt1
        | EMPTY
```

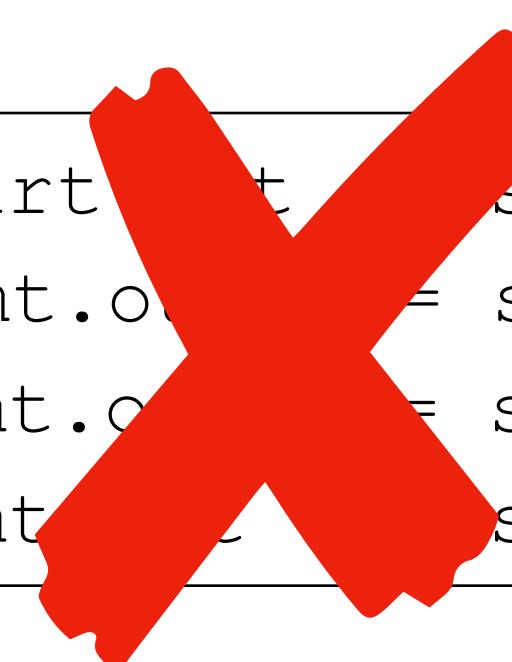
```
stmt.in  := 0           , start.out := stmt.out , start.ok := stmt.ok
stmt1.in := stmt.in + 1, stmt.out  := stmt1.out , stmt.ok := stmt.out ≥ 0
stmt1.in := stmt.in - 1, stmt.out  := stmt1.out , stmt.ok := stmt.out ≥ 0
                           stmt.out  := stmt.in , stmt.ok := stmt.out = 0
```

# 속성이 주도하는 제약 생성

## 다시, 괄호 언어

```
start ::= stmt
stmt  ::= LPAR stmt1
       | RPAR stmt1
       | EMPTY
```

```
stmt.in  := 0           , start.out = stmt.out , start.ok := stmt.ok
stmt1.in := stmt.in + 1, stmt.out = stmt1.out , stmt.ok := stmt.out ≥ 0
stmt1.in := stmt.in - 1, stmt.out = stmt1.out , stmt.ok := stmt.out ≥ 0
stmt1.in := stmt.in ,   stmt.out = 0          , stmt.ok := stmt.out = 0
```



```
start ::= stmt
stmt  ::= LPAR stmt1
       | RPAR stmt1
       | EMPTY
```

```
RPAR.check := True
EMPTY.check := True
```

# 속성이 주도하는 제약 생성

## 다시, 괄호 언어

```
start ::= stmt
stmt  ::= LPAR stmt1
       | RPAR stmt1
       | EMPTY
```

```
stmt.in  := 0           , start.in = stmt.out , start.ok := stmt.ok
stmt1.in := stmt.in + 1, stmt.out = stmt1.out , stmt.ok := stmt.out ≥ 0
stmt1.in := stmt.in - 1, stmt.out = stmt1.out , stmt.ok := stmt.out ≥ 0
stmt1.in := stmt.in ,   stmt.out = 0
```

```
start ::= stmt
stmt  ::= LPAR stmt1
       | RPAR stmt1
       | EMPTY
```

```
RPAR.check := True
EMPTY.check := True
```

언제, 어디에 분석?  
속성이 check가 True인 문법 기호

# 속성이 주도하는 제약 생성

## 다시, 괄호 언어

```
start ::= stmt
stmt  ::= LPAR stmt1
       | RPAR stmt1
       | EMPTY
```

```
stmt.in  := 0           , start.in = stmt.out , start.ok := stmt.ok
stmt1.in := stmt.in + 1, stmt.out = stmt1.out , stmt.ok := stmt.out ≥ 0
stmt1.in := stmt.in - 1, stmt.out = stmt1.out , stmt.ok := stmt.out ≥ 0
stmt1.in := stmt.in ,   stmt.out = 0
```

```
start ::= stmt
stmt  ::= LPAR stmt1
       | RPAR stmt1
       | EMPTY
```

```
RPAR.check := True
EMPTY.check := True
```

언제, 어디에 분석?  
속성이 check가 True인 문법 기호

어떻게 분석?  
이미 존재하는 분석기(괄호 판단 알고리즘) 이용

# 속성이 주도하는 제약 생성

점진적(incremental) 파서로 모두 고려하기

< start : • stmt | Ø >

파서 스택

읽은 문자열 없음

# 속성이 주도하는 제약 생성

점진적(incremental) 파서로 모두 고려하기

- 일반 파서: 더 이상 읽을 문자가 없음

그러면 \$ (=EndOfFile)인가 보다!

< start : stmt • | start.out := 0 >

파서 스택

파싱 완료!

# 속성이 주도하는 제약 생성

점진적(incremental) 파서로 모두 고려하기

- 일반 파서: 더 이상 읽을 문자가 없음

그러면 \$ (=EndOfFile)인가 보다!

- LLM은 모든 코드를 한 번에 만들지 않고 여러 차례에 걸쳐 나누어 만듦

```
< start : stmt • | start.out := 0 >
```

파서 스택

파싱 완료!

# 속성이 주도하는 제약 생성

## 점진적(incremental) 파서로 모두 고려하기

- 일반 파서: 더 이상 읽을 문자가 없음

그러면 \$ (=EndOfFile)인가 보다!

- LLM은 모든 코드를 한 번에 만들지 않고 여러 차례에 걸쳐 나누어 만듦

```
< start : stmt • | start.out := 0 >
```

파서 스택

파싱 완료!

읽을 문자열이 없어도 \$로 취급하지 않고 가능한 모든 경우의 수를 고려해주어야 함!

# 속성이 주도하는 제약 생성

점진적(incremental) 파서로 모두 고려하기

- 점진적 파서: 가능한 입력에 따른 파서 상태를 모두 고려

< start : • stmt | Ø >

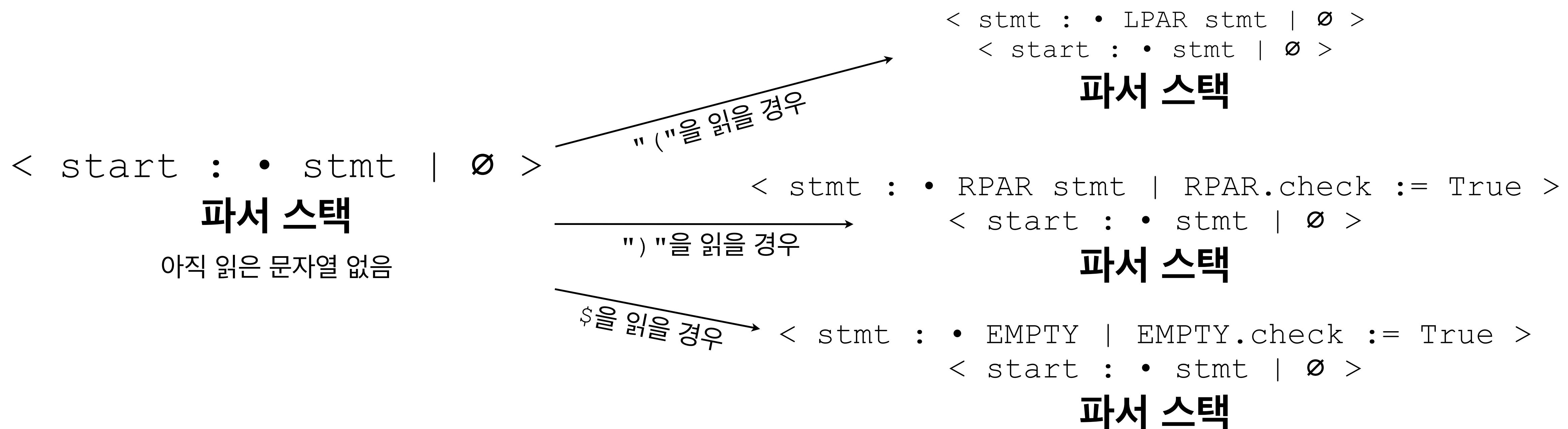
**파서 스택**

아직 읽은 문자열 없음

# 속성이 주도하는 제약 생성

점진적(incremental) 파서로 모두 고려하기

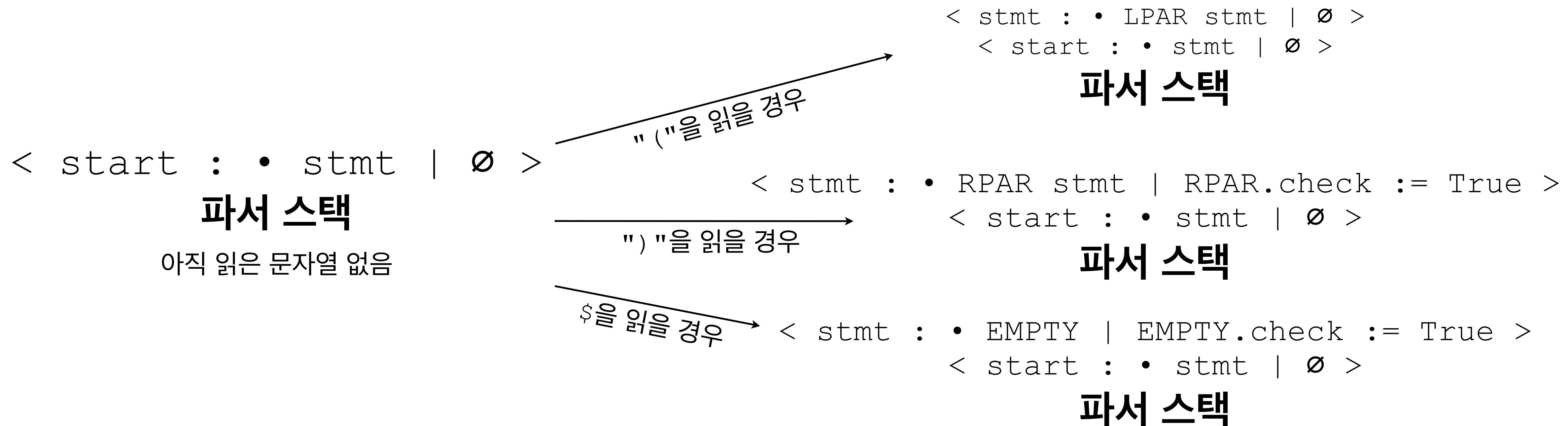
- 점진적 파서: 가능한 입력에 따른 파서 상태를 모두 고려



# 속성이 주도하는 제약 생성

점진적(incremental) 파서로 모두 고려하기

- 점진적 파서: 가능한 입력에 따른 파서 상태를 모두 고려
- 올바른 입력: 각 파서 상태를 순회하며 의미적 올바름을 지키는 파서 상태가 하나라도 존재



# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\phi$ )*

```
|1:  
|2:  
|3:  
|4:  
|5:  
|6:  
|7:  
|8:  
|9:  
|10:  
|11:  
|12:  
|13:  
|14:  
|15:
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\varphi$ )*

```
1:  
2:  
3:  
4:  
5:  
6:  
7:  
8:  
9:  
10:  
11:  
12:  
13:  
14:  
15:
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\phi$ )*

### 초기화

1:	$\text{싹수} \leftarrow T$	
2:	$\bar{\Gamma} \leftarrow \{[\text{초기파서스택}]\}$	: 현재 가능한 파서 스택 집합
3:	$\sigma \leftarrow \phi$	: 등장한 문법 기호와 그 기호의 속성 정보
4:		
5:		
6:		
7:		
8:		
9:		
10:		
11:		
12:		
13:		
14:		
15:		

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\varphi$ )*

```
I: 싹수  $\leftarrow T$ 
2:  $\bar{\Gamma} \leftarrow \{[\text{초기파서스택}]\}$ 
3:  $\sigma \leftarrow \phi$ 
4: for 글자  $c \in P$  do
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\phi$ )*

```
I: 썩수 ← T
2:  $\bar{\Gamma} \leftarrow \{[\text{초기파서스택}]\}$ 
3:  $\sigma \leftarrow \phi$ 
4: for 글자  $c \in P$  do
5:    $\bar{\Gamma} \leftarrow \text{모든가능한다음스택}(c, \bar{\Gamma})$  : 입력 기호  $c$ 를 받아 도달 가능한 모든 파서 스택
6:   c의싹수 ← T
7:
8:
9:
10:
11:
12:
13:
14:
15:
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\varphi$ )*

```
I: 썩수 ← T
2:  $\bar{\Gamma} \leftarrow \{[\text{초기파서스택}]\}$ 
3:  $\sigma \leftarrow \phi$ 
4: for 글자  $c \in P$  do
5:    $\bar{\Gamma} \leftarrow \text{모든가능한다음스택}(c, \bar{\Gamma})$ 
6:   c의싹수 ← T
7:   for  $\Gamma \in \bar{\Gamma}$  do
8:
9:
10:
11:
12:
13:
14:
15:
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\varphi$ )*

```
I:    짹수 ← T
2:     $\bar{\Gamma} \leftarrow \{[\text{초기파서스택}]\}$ 
3:     $\sigma \leftarrow \phi$ 
4:    for 글자  $c \in P$  do
5:         $\bar{\Gamma} \leftarrow \text{모든가능한다음스택}(c, \bar{\Gamma})$ 
6:        c의 짹수 ← T
7:        for  $\Gamma \in \bar{\Gamma}$  do
8:             $\Gamma$ 의 짹수 ← T
9:             $\bar{a} \leftarrow \text{속성얻기}(\Gamma)$  : 해당 파서 스택에 등장하는 기호들의 속성 정보 얻기
10:            $\sigma \leftarrow \sigma \oplus \bar{a}$       : 이전까지 수집한 기호들의 속성 정보에 덮어쓰기
11:
12:
13:
14:
15:
```

속성 정보 업데이트

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\varphi$ )*

```
I:  짹수 ← T
2:   $\bar{\Gamma}$  ← {[초기파서스택]}
3:   $\sigma \leftarrow \phi$ 
4:  for 글자  $c \in P$  do
5:     $\bar{\Gamma} \leftarrow$  모든가능한다음스택( $c, \bar{\Gamma}$ )
6:     $c$ 의 짹수 ← T
7:    for  $\Gamma \in \bar{\Gamma}$  do
8:       $\Gamma$ 의 짹수 ← T
9:       $\bar{a} \leftarrow$  속성얻기( $\Gamma$ )
10:      $\sigma \leftarrow \sigma \oplus \bar{a}$ 
11:     for  $X \in$  체크할기호( $\sigma, \bar{a}$ ) do : 체크해야할 기호들을 파악
12:
13:
14:
15:
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\varphi$ )*

```
I:  짹수 ← T
2:   $\bar{\Gamma}$  ← {[초기파서스택]}
3:   $\sigma$  ←  $\phi$ 
4:  for 글자  $c \in P$  do
5:     $\bar{\Gamma}$  ← 모든가능한다음스택( $c, \bar{\Gamma}$ )
6:     $c$ 의 짹수 ← T
7:    for  $\Gamma \in \bar{\Gamma}$  do
8:       $\Gamma$ 의 짹수 ← T
9:       $\bar{a}$  ← 속성얻기( $\Gamma$ )
10:      $\sigma$  ←  $\sigma \oplus \bar{a}$ 
11:     for  $X \in \text{체크할기호}(\sigma, \bar{a})$  do
12:        $\Gamma$ 의 짹수 ←  $\Gamma$ 의 짹수  $\wedge$  분석기( $P, \Gamma, \sigma, X$ ) : 해당 기호의 위치에서만 분석기 작동
13:
14:
15:
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\varphi$ )*

```
I: 썩수 ← T
2:  $\bar{\Gamma} \leftarrow \{[\text{초기파서스택}]\}$ 
3:  $\sigma \leftarrow \phi$ 
4: for 글자  $c \in P$  do
5:    $\bar{\Gamma} \leftarrow \text{모든가능한다음스택}(c, \bar{\Gamma})$ 
6:   c의싹수 ← T
7:   for  $\Gamma \in \bar{\Gamma}$  do
8:      $\Gamma$ 의싹수 ← T
9:      $\bar{a} \leftarrow \text{속성얻기}(\Gamma)$ 
10:     $\sigma \leftarrow \sigma \oplus \bar{a}$ 
11:    for  $X \in \text{체크할기호}(\sigma, \bar{a})$  do
12:       $\Gamma$ 의싹수 ←  $\Gamma$ 의싹수  $\wedge$  분석기( $P, \Gamma, \sigma, X$ ) : 모든 기호가 성질을 만족하면 TRUE
13:
14:
15:
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\phi$ )*

```
I:  짹수 ← T
2:   $\bar{\Gamma}$  ← {[초기파서스택]}
3:   $\sigma$  ←  $\phi$ 
4:  for 글자  $c \in P$  do
5:     $\bar{\Gamma}$  ← 모든가능한다음스택( $c, \bar{\Gamma}$ )
6:     $c$ 의 짹수 ← T
7:    for  $\Gamma \in \bar{\Gamma}$  do
8:       $\Gamma$ 의 짹수 ← T
9:       $\bar{a}$  ← 속성얻기( $\Gamma$ )
10:      $\sigma$  ←  $\sigma \oplus \bar{a}$ 
11:     for  $X \in \text{체크할기호}(\sigma, \bar{a})$  do
12:        $\Gamma$ 의 짹수 ←  $\Gamma$ 의 짹수  $\wedge$  분석기( $P, \Gamma, \sigma, X$ )
13:        $c$ 의 짹수 ←  $c$ 의 짹수  $\vee$   $\Gamma$ 의 짹수 : 파서 스택 중 어느 하나라도 만족하면 TRUE
14:
15:
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\phi$ )*

```
I:    짝수 ← T
2:     $\bar{\Gamma} \leftarrow \{[\text{초기파서스택}]\}$ 
3:     $\sigma \leftarrow \phi$ 
4:    for 글자  $c \in P$  do
5:         $\bar{\Gamma} \leftarrow \text{모든가능한다음스택}(c, \bar{\Gamma})$ 
6:         $c\text{의짝수} \leftarrow T$ 
7:        for  $\Gamma \in \bar{\Gamma}$  do
8:             $\Gamma\text{의짝수} \leftarrow T$ 
9:             $\bar{a} \leftarrow \text{속성얻기}(\Gamma)$ 
10:            $\sigma \leftarrow \sigma \oplus \bar{a}$ 
11:           for  $X \in \text{체크할기호}(\sigma, \bar{a})$  do
12:                $\Gamma\text{의짝수} \leftarrow \Gamma\text{의짝수} \wedge \text{분석기}(P, \Gamma, \sigma, X)$ 
13:                $c\text{의짝수} \leftarrow c\text{의짝수} \vee \Gamma\text{의짝수}$ 
14:               짝수 ← 짝수  $\wedge c\text{의짝수.}$  : 모든 문자  $c$ 에 대해 만족하면 TRUE
15: return 짝수
```

# 언제, 어디를 검사할 것인가?

## 미완성 프로그램 검사 알고리즘

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\phi$ )*

```
I: 짝수  $\leftarrow T$ 
2:  $\bar{\Gamma} \leftarrow \{[\text{초기파서스택}]\}$ 
3:  $\sigma \leftarrow \phi$ 
4: for 글자  $c \in P$  do
5:    $\bar{\Gamma} \leftarrow \text{모든가능한다음스택}(c, \bar{\Gamma})$ 
6:    $c\text{의 짝수} \leftarrow T$ 
7:   for  $\Gamma \in \bar{\Gamma}$  do
8:      $\Gamma\text{의 짝수} \leftarrow T$ 
9:      $\bar{a} \leftarrow \text{속성얻기}(\Gamma)$ 
10:     $\sigma \leftarrow \sigma \oplus \bar{a}$ 
11:    for  $X \in \text{체크할기호}(\sigma, \bar{a})$  do
12:       $\Gamma\text{의 짝수} \leftarrow \Gamma\text{의 짝수} \wedge \text{분석기}(P, \Gamma, \sigma, X)$ 
13:     $c\text{의 짝수} \leftarrow c\text{의 짝수} \vee \Gamma\text{의 짝수}$ 
14:  짝수  $\leftarrow \text{짝수} \wedge c\text{의 짝수}$ .
15: return 짝수
```

언제? 검사해야 할 기호가 존재할 때만

어디? 해당 기호의 그 위치만

# 변수명 올바름을 보장해보자

## 변수명 올바름을 위한 준비물

*IsValid(미완성프로그램  $P$ , 속성문법  $AG$ , 정적분석기  $A_\phi$ )*

```
I:    징수 ← T
2:     $\bar{\Gamma} \leftarrow \{[\text{초기파서스택}]\}$ 
3:     $\sigma \leftarrow \phi$ 
4:    for 글자  $c \in P$  do
5:         $\bar{\Gamma} \leftarrow \text{모든가능한다음스택}(c, \bar{\Gamma})$ 
6:         $c\text{의 징수} \leftarrow T$ 
7:        for  $\Gamma \in \bar{\Gamma}$  do
8:             $\Gamma\text{의 징수} \leftarrow T$ 
9:             $\bar{a} \leftarrow \text{속성얻기}(\Gamma)$ 
10:            $\sigma \leftarrow \sigma \oplus \bar{a}$ 
11:           for  $X \in \text{체크할기호}(\sigma, \bar{a})$  do
12:                $\Gamma\text{의 징수} \leftarrow \Gamma\text{의 징수} \wedge \text{분석기}(P, \Gamma, \sigma, X)$ 
13:                $c\text{의 징수} \leftarrow c\text{의 징수} \vee \Gamma\text{의 징수}$ 
14:           징수 ← 징수  $\wedge c\text{의 징수}.$ 
15:       return 징수
```

## 준비물

- I. 속성 문법:
  - 변수 정의/참조
  - 정의-참조 역전 구간
2. 체크할기호 정의
3. 변수명 오사용 판단 분석기

# 변수명 올바름을 보장해보자

## 속성 문법

- *bnd* : 참조로 사용되는 변수라면 True, 정의라면 False

# 변수명 올바름을 보장해보자

## 속성 문법

- $bnd$  : 참조로 사용되는 변수라면 True, 정의라면 False

Ex)

Assign → Name := Expr

Name.bnd = F, Expr.bnd = T

# 변수명 올바름을 보장해보자

## 속성 문법

- $bnd$  : 참조로 사용되는 변수라면 True, 선언이라면 False

Ex)

Assign → Name := Expr

Name.bnd = F, Expr.bnd = T

- $ctx$  : 선언되지 않은 변수의 참조가 먼저 일어나고(U), 나중에 선언됨(R)

# 변수명 올바름을 보장해보자

## 속성 문법

- *bnd* : 참조로 사용되는 변수라면 True, 선언이라면 False

Ex)

Assign → Name := Expr

Name.bnd = F, Expr.bnd = T

- *ctx* : 선언되지 않은 변수의 참조가 먼저 일어나고(U), 나중에 선언됨(R)

Ex)

Comp → Expr Fors

Expr.ctx = Comp.ctx · U

[ i for i in range(5) ]

Fors.ctx = Comp.ctx · R

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

1. 참조를 위한 변수  $\wedge$  참조-정의 역전 구간이 아님

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 1. 참조를 위한 변수 $\wedge$ 참조-정의 역전 구간이 아님

미완성 코드

app = 1

ban

파서 스택

1. [..., < assign : Name • := Expr | Name.bnd  $\mapsto$  False >]

2. [..., < Expr : Name • + Name | Name.bnd  $\mapsto$  True >]

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 1. 참조를 위한 변수 $\wedge$ 참조-정의 역전 구간이 아님

미완성 코드

app = 1

ban

파서 스택

1. [..., < assign : Name • := Expr | Name.bnd  $\mapsto$  False >]

2. [..., < Expr : Name • + Name | Name.bnd  $\mapsto$  True >]

아직은 가능한 파서 스택에 참조/정의 모두 가능하기 때문에 검사 대상에 포함되지 않음

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 1. 참조를 위한 변수 $\wedge$ 참조-정의 역전 구간이 아님

미완성 코드

```
app = 1  
ban +
```

파서 스택

1. [..., < Expr : Name + • Name | Name.bnd  $\mapsto$  True >]

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 1. 참조를 위한 변수 $\wedge$ 참조-정의 역전 구간이 아님

미완성 코드

```
app = 1  
ban +
```

파서 스택

```
1. [..., < Expr : Name + • Name | Name.bnd  $\mapsto$  True >]
```

ban은 **참조**를 위한 변수로 체크해야 할 기호

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 1. 참조를 위한 변수 ∧ 참조-정의 역전 구간이 아님

미완성 코드

```
app = 1  
ban +
```

파서 스택

```
1. [..., < Expr : Name + • Name | Name.bnd ↣ True >]
```

ban은 **참조**를 위한 변수로 체크해야할 기호

변수명 위치에서 사용 가능한 변수 집합을 주는 정적 분석기 이용

- 변수명이 더 생성될 수 있는 코드의 마지막 부분인 경우: prefix만 검사
- 그렇지 않은 경우: 가능한 변수 집합 중 하나인지 검사

가능한 변수명(app)에 해당하지 않음: 의미적 올바름 X

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 2. 참조를 위한 변수 $\wedge$ 참조-정의 역전 구간

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 2. 참조를 위한 변수 $\wedge$ 참조-정의 역전 구간

- 역전 구간에서는 참조로 사용되는 변수가 등장해도 즉시 검사하지 않음
- 역전 구간이 끝난 후, 더 이상 정의가 불가능해지는 시점에 검사

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 2. 참조를 위한 변수 $\wedge$ 참조-정의 역전 구간

- 역전 구간에서는 참조로 사용되는 변수가 등장해도 즉시 검사하지 않음
- 역전 구간이 끝난 후, 더 이상 정의가 불가능해지는 시점에 검사

```
[ i for j in [ k for t in [ x for y in range(5)
```

# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 2. 참조를 위한 변수 $\wedge$ 참조-정의 역전 구간

- 역전 구간에서는 참조로 사용되는 변수가 등장해도 즉시 검사하지 않음
- 역전 구간이 끝난 후, 더 이상 정의가 불가능해지는 시점에 검사

[ i for j in [ k for t in [ x for y in range(5)



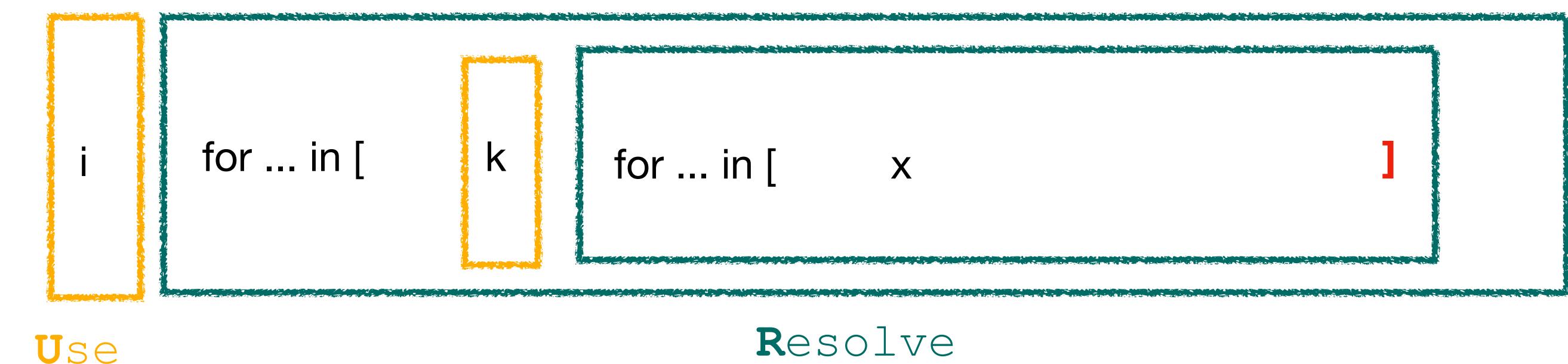
# 변수명 올바름을 보장해보자

체크할기호 + 정적분석기

## 2. 참조를 위한 변수 $\wedge$ 참조-정의 역전 구간

- 역전 구간에서는 참조로 사용되는 변수가 등장해도 즉시 검사하지 않음
- 역전 구간이 끝난 후, 더 이상 정의가 불가능해지는 시점에 검사

[ i for j in [ k for t in [ x for y in range(5) ] ] ]



x의 역전 구간이 종료되어 분석하고, i와 k는 검사할 필요 없음  
x는 정의된 변수명이 아니므로 역전 구간을 종료시키는 ]을 거부

# 실험

## 실험 설정

- **Baseline:** Reject Sampling (5 turns)  
변수명 오류가 있으면 피드백과 함께 최대 5번 재생성
- **Benchmark:** 문법 올바름만 보장하는 제약 생성시 변수명 오사용을 포함한 케이스  
MBPP+HumanEval : Python, OCaml  
BigCodeBench : Python
- **Parser:** Lark 라이브러리 (+ 속성 계산하도록 변형)
- **Static Analysis Tool:** Pyflake (Python), Merlin(OCaml)
- **Metrics:**
  1. 변수 사용이 올바른 코드 완성 비율
  2. 토큰 하나당 소모된 추가 비용

$$\frac{\text{전체 생성 시간} - \text{문법 올바름 보장에 걸린 추가 시간}}{\text{성공한 시도에서 생성한 토큰 수}}$$

# 실험

## OCaml

Model	Success rate		Per-token time (Avg. in milliseconds)		
	RESAMPLE	GENCOR	RESAMPLE	GENCOR	
HumanEval	CodeLlama-7B	38.5% (10/26)	<b>88.5% (23/26)</b>	214.3	<b>77.1</b>
	CodeLlama-34B	45.5% (10/22)	<b>86.4% (19/22)</b>	133.5	<b>74.8</b>
	DSCoder-6.7B	35.7% (10/28)	<b>78.6% (22/28)</b>	110.8	<b>102.7</b>
	DSCoder-33B	22.2% (1/9)	<b>88.9% (8/9)</b>	752.9	<b>226.3</b>
	Qwen2.5-7B	74.1% (20/27)	<b>74.1% (20/27)</b>	62.1	<b>42.1</b>
	Qwen2.5-32B	58.8% (9/16)	<b>87.5% (14/16)</b>	147.2	<b>116.9</b>
MBPP	CodeLlama-7B	60.8% (14/23)	<b>100% (23/23)</b>	82.6	<b>39.7</b>
	CodeLlama-34B	66.7% (10/15)	<b>93.3% (14/15)</b>	119.1	<b>94.8</b>
	DSCoder-6.7B	40% (10/25)	<b>76% (19/25)</b>	68.0	<b>61.2</b>
	DSCoder-33B	36.4% (4/11)	<b>90.9% (10/11)</b>	140.8	<b>90.3</b>
	Qwen2.5-7B	76.9% (20/26)	<b>92.3% (24/26)</b>	68.1	<b>40.5</b>
	Qwen2.5-32B	<b>81.3% (13/16)</b>	<b>81.3% (13/16)</b>	141.0	<b>103.0</b>

높은 성공률 달성

더 작은 추가 시간 비용

# 실험

## Python

Model	Success rate		Per-token time (Avg. in milliseconds)	
	ReSample	GenCor	ReSample	GenCor
BigCodeBench	CodeLlama-7B	9.2% (8/87)	71.3% (62/87)	102.1
	CodeLlama-34B	29.6% (16/54)	63.0% (34/54)	418.5
	DSCoder-6.7B	48.3% (14/29)	41.4% (12/29)	80.2
	DSCoder-33B	69.2% (18/26)	57.7% (15/26)	486.7
	Qwen2.5-7B	66.7% (30/45)	71.1% (32/45)	58.2
	Qwen2.5-32B	66.6% (2/3)	100% (3/3)	615.2
MBPP HumanEval +	CodeLlama-7B	40% (8/20)	35% (7/20)	63.2
	CodeLlama-34B	92% (23/25)	72% (18/25)	317.7
	DSCoder-6.7B	100% (7/7)	57% (4/7)	60.5
	DSCoder-33B	—	—	—
	Qwen2.5-7B	100% (2/2)	100% (2/2)	63.2
	Qwen2.5-32B	—	—	—

높은 성공률 달성

더 적은 추가 시간 비용

# Thank you

## 속성이 주도하는 제약 생성으로 의미 지키기: 변수명 오용 방지

Attribute-Guided Semantic Filtering for Constrained Decoding with an Application to Variable Misuse Prevention

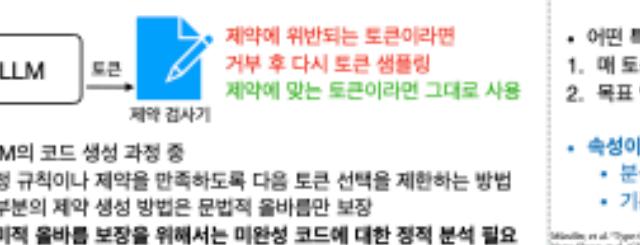
박준성, 김진상, 이우석

한양대학교 프로그래밍 시스템 연구실 (PSL)



### 문제

#### 제약 생성(Constrained Decoding)



- 제약의 코드 생성 과정 중
- 특정 규칙이나 제약을 만족하도록 다음 토큰 선택을 제한하는 방법
- 대부분의 제약 생성 방법은 문법적 올바름만 보장
- 의미적 올바름 보장을 위해서는 미완성 코드에 대한 정적 분석 필요

#### 기존 의미 제약 생성 방법의 한계

- 어떤 특성을 언제, 어떻게 분석할지가 하나로 묶쳐져 있음
- 예. 도구 생성마다 지금까지 만든 코드 전체 분석
- 목표 언어+특성을 위한 분석기 직접 만들

- 속성이 주도하는 제약 생성!
- 분석이 언제, 어디에 필요한지는 속성을 통해 판단하고,
- 기존의 분석기를 활용하여 의미적 올바름 보장

Moulier et al. "Open-Source Code Generation with Language Model PLDG" (2023).  
Moulier et al. "A Step-by-Step Argumentative Framework for Semantically Constraining the Output of Language Models" (2023).

#### 왜 분석기만 이용하면 안될까?

- 코드 일부 부분만 보고는 올바름 판단을 위한  
역학이 불충분한 경우 존재
- ex. Python - 변수명 오류 판단 분석기

- [i]
- 1. [i]
- 2. [i for i in range(10)]

- 2번 코드를 생성하면 변수명 오류 알지만  
안전한 분석기는 i의 생성을 막아야 함

### 방법

#### 속성달린 문법 (Attribute Grammars): 꽂호 인여

생김새 규칙	의미 규칙
start → p	p.i := 0, start.o := p.o
→ LPAR pi	p.i := p.i + 1, p.o := p.i.o
RPAR pi	p.i := p.i - 1, p.o := p.i.o
EMPTY	p.o := p.i

- 속성: 문법 기호에 값을 저장할 수 있는 변수 (X.a : 문법 기호 X의 속성 a)
- #: 지금까지 닫히지 않은 괄호 (상속) o: 아직도 닫히지 않은 괄호 (상속)

#### 파싱을 통해 결정하는 속성의 값 [ex. ( ) \$ ]

자식 규칙
< p : E *   p.i > -1 > =0-1
< p : R * p   p.i > 0 > =1-1
< p : R * p   p.i > 1 > =0+1
< p : L * p   p.i > 0 > =0
< start : p *   p.o > -4 > =-1

- 모든 입력을 읽은 파서 상태

다음 입력: \$ = <EOS>

reduction이 일어날 때 험에 있는 속성은 같이 사라짐

#### 그럼 속성만 있으면 다 되는건가?

추가 의미 규칙	속성이 주도하는 제약 생성
외부적 올바름: 모든 괄호쌍이 균형잡힌 상태	start → p ... start.ok := p.ok P → LPAR pi ... p.ok := p.o.x   RPAR pi ... p.ok := p.o.z   EMPTY ... p.ok := p.o.D

- 속성과 의미 규칙을 잘 정의하면, 원하는 의미적 올바름을 보장할 수 있음 • 언제, 어디에 분석할지는 속성이 알려주고,
- 그러나 복잡한 언어에 대해서는 잘 정의하기 어렵고, ex. 속성이 check인 기호가 있다면 분석 이미 존재하는 정적 분석기를 다시 만드는 것과 다르지 않음

### 적용 (변수명 오용 방지)

#### 속성 문법 정의

언제, 어디로 분석해야 할까?
• bnd: 참조로 사용되는 이름이라면 True, 선언이라면 False
Ex) Assign → Name := Expr Name.bnd = F Expr.bnd = T Expr.ctx = E
• ctx: 선언되지 않은 변수의 참조가 먼저 일어나고(U), 나중에 선언됨(P)
Ex) Comp → Expr Fors Exp.ctx = Comp.ctx.U [ i for i in range(5) ] Fors.ctx = Comp.ctx.P

#### 언제, 어디로 분석해야 할까?

SYMBOLS_TO_CHECK( $\sigma, \bar{\alpha}$ ) =
$\{X \in N_{\sigma} \mid \sigma(X.bnd) = \text{U} \wedge (\sigma(X.ctx) = \text{v} \vee$
$\exists Y \in N, s[i..min(j+1, s[i] - 1)..k \leq s[i].ctx]$
where $s = \sigma(X.ctx)\}$
$\sigma \leftarrow \sigma @ \{X, a \mapsto \bar{\alpha}[X], a \in A(X)\}$

1. ..., < p : \* L P | Ø >

2. ..., < p : \* R P | R.check = True >

3. ..., < p : \* E | E.check = True >

#### 참조-정의 역전 구간

- 역전 구간에서는 참조로 사용되는 변수가 등장해도 즉시 검사하지 않음
- 역전 구간이 끝난 후, 더 이상 정의가 불가능해지는 시점에 검사

Use	Resolve
[ i for j in [ k for t in [ x for y in range(5) ] ] ]	[ i for ... in [ k for ... in [ x ] ] ]

x의 역전 구간이 종료되어 분석하고, i와 k는 검사할 필요 없음

x는 정의된 변수명이 아니므로 역전 구간을 종료시키는 ]을 거부

### 실험

#### Baseline: Reject Sampling (5 turns)

- 변수명 오류가 있으면 피드백과 함께 최대 5번 패生育
- Benchmark: 문법 올바름만 보장하는 제약 생성
- 언어: Python, OCaml, BigCodeBench
- 파서: Lark 라이브러리 (+ 속성 계산하도록 변경)
- 정적 분석 도구: Pyflakes (Python), Merlin (OCaml)
- 메트릭: 1. 변수 사용이 올바른 코드 완성 비율  
2. 트랜잭션 처리 소모된 추가 비용  
전체 성능 시간 / 문법 올바름 보장에 걸린 모비虑

#### 고정 시험: CodeLlama34B, BigCodeBench/5

Model	Success rate		Per-token time (Avg. in milliseconds)	
	RefBaseline	GenCom	RefBaseline	GenCom
CodeLlama-7B	15.0% (16/107)	64.8% (48/74)	94.5	62.6
CodeLlama-13B	15.0% (16/107)	64.8% (48/74)	94.5	62.7
CodeLlama-34B	64.9% (21/33)	64.4% (14/22)	76.4	57.8
BigCodeBench-34B	69.3% (18/26)	57.7% (15/26)	66.7	187.8
BigCodeBench-51B	69.1% (18/27)	58.3% (16/27)	58.5	28.1
BigCodeBench-77B	66.4% (21/31)	50.0% (8/31)	65.2	243.8
CodeLlama-7B	38.3% (10/26)	66.0% (23/36)	79.5	61.6
CodeLlama-13B	34.1% (10/27)	66.2% (23/37)	126.5	62.5
CodeLlama-34B	37.4% (13/35)	71.4% (25/35)	74.8	54.8
BigCodeBench-34B	27.5% (7/26)	66.3% (14/26)	125.4	55.6
BigCodeBench-51B	27.5% (7/26)	66.3% (14/26)	64.5	43.8
BigCodeBench-77B	27.5% (7/26)	66.4% (22/32)	125.8	111.7

Copyright 2028, PSL, Hanyang University. All rights reserved.

돌은 성공률 달성 디 족은 추가 시간 비용