

# 썩수 분석으로 프로그램 합성 보조하기

썩수 분석의 효을 끌어올리기

# 요약

- 배경
- 합성 잘하기 -> 부품 빨리 늘리기
  - 합성이 더 잘 됨 (경험적)
  - 싹수 분석 성공의 효과 큼
- 분석 잘하기 -> 중간 결과 재활용으로 싹수 분석 효율 높이기
  - 후보에 구멍이 많아야 재활용 효과 큼
- 희망은 어디에?
  - 부품 빨리 늘리기 <> 구멍 많은 후보 만들기

배경

# 양방향 합성

$$\begin{array}{lcl}
E & \rightarrow & 0 \mid 1 \mid -1 \\
& | & x \\
& | & E + E \\
& | & E - E \\
& | & E \wedge E \\
& | & E \vee E \\
& | & E \oplus E \\
& | & E \gg E \\
& | & []
\end{array}$$

Top  
Down  
후보 나열


$$[\ ] + [\ ]$$

[ ]

$$[\ ]^{\wedge}[\ ]$$

[ ] >> [ ]

정답:

$$((x + 1) \wedge x) \gg 1$$

Bottom  
Up  
부품 생산



0

1

-1



# 양방향 합성

$$\begin{array}{lcl}
E & \rightarrow & 0 \mid 1 \mid -1 \\
& | & x \\
& | & E + E \\
& | & E - E \\
& | & E \wedge E \\
& | & E \vee E \\
& | & E \oplus E \\
& | & E \gg E \\
& | & []
\end{array}$$

Top  
Down  
후보 나열


$$\begin{aligned} & [] + [] \\ & ([ ] + [ ]) + [ ] \\ & ([ ] - [ ]) + [ ] \\ & \dots \end{aligned}$$
$$\begin{aligned} & [] \\ & []^{}[] \\ & ( [] \& [] )^{}[] \\ & ( [] | [] )^{}[] \\ & \dots \end{aligned}$$
$$\begin{aligned} & [] \gg [] \\ & ([[] | []]) \gg [] \\ & ([[] ^ []]) \gg [] \\ & \dots \end{aligned}$$

정답:  
 $((x + 1) \wedge x) \gg 1$

Bottom  
Up  
부품 생산

x&1

 $x^1$ 

**x-1**

**X+X**

**x+1**

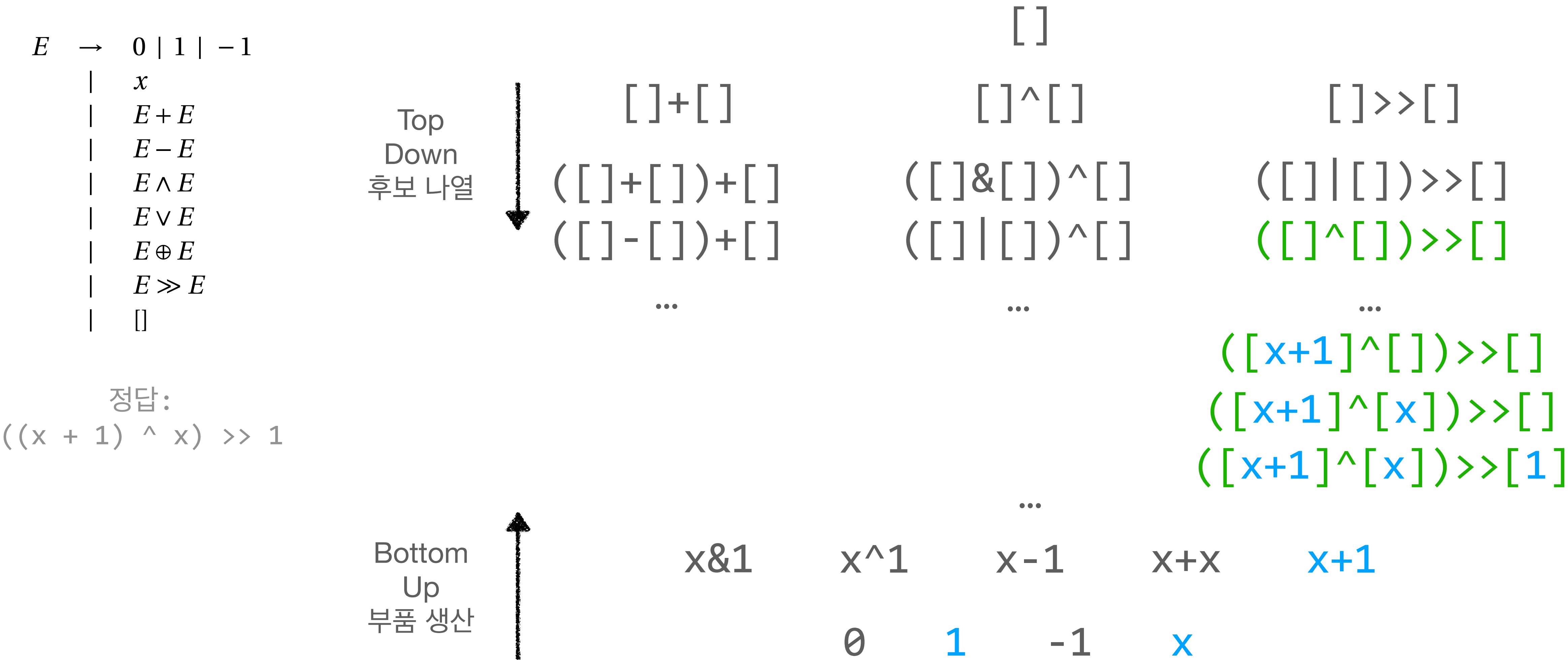
0

1

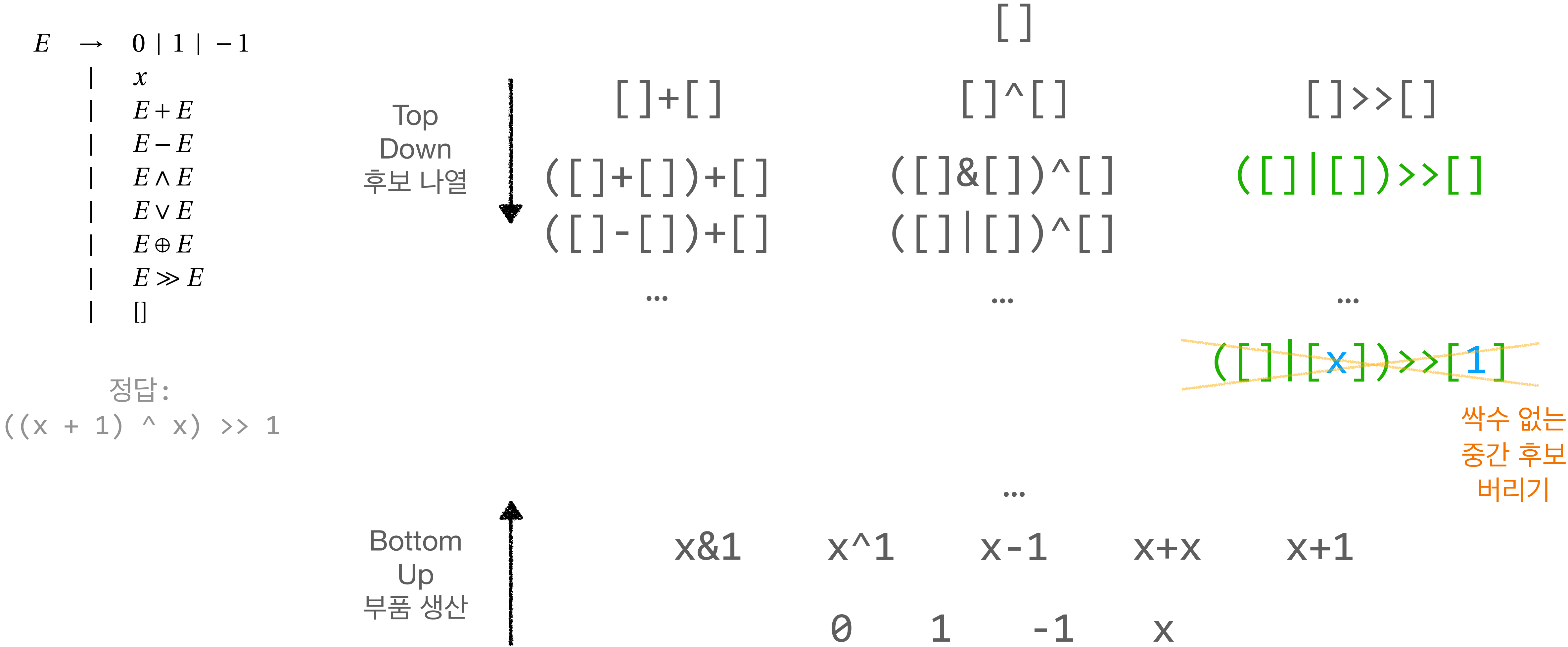
-1

X

# 양방향 합성



# 양방향 합성 + 싹수 분석



# 씩수 분석의 역할

- 합성 중 만들어보는 후보의 “순서”는 보존 => 합성 최종 결과는 같다
- 합성 중 불필요하게 만들어보는(씩수 없는) 후보를 건너뛰는 것
- 남은 빈칸과 전체 부품 많을 때 효과 큼 (부품 수 \* 빈칸 수)

단순 합성

XXX0

+씩수 분석

XXXX \_ XXX \_ XXXX \_ XXXXXX \_ XXXXXXXX \_ XXXX0

효과  
적은

효과  
크



# 주요 측정 대상

- 합성 성능
  - [가장 중요!] 합성 시간
  - 만든 부품 수 (~ 부품의 최대 크기)
  - 만든 중간 후보 수 (~ 중간 후보의 최대 빈칸 수) & 만든 모든 후보 수
- 분석 효율
  - 분석 시간
  - 분석으로 덜 만들게 된 후보 수

합성 잘 하기

# 합성 자체를 충분히 잘 해야 분석도 가치 있음

- 싹수 분석은 합성 중 불필요한 삽질을 덜어줄 뿐
- 해야할 삽질이 지나치게 많으면
  - 합성 성능이 나뉨
- 쓸데없이 삽질을 많이 하면서 그 삽질을 많이 줄여 싹수 분석의 효과를 과대평가하면
  - 기만

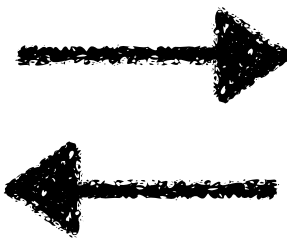
# 양방향 합성 알고리즘

<입출력 기반 합성기>

입력: 입출력 쌍

출력: 합성해본 프로그램

- 1. 부품 세트를 만든다
- 2. 후보 세트를 만든다
- 3. 후보 하나를 고른다  
(씩수 분석 시도)
  - 3-1. 각 빈칸마다 가능한 모든 부품을 하나씩 끼워본다  
(끼울 때마다씩수 분석 시도)
    - 3-1-1. 빈칸을 다 채우면 입출력 쌍을 만족하는지 검사한다  
만족하면 출력
- 4. 후보가 다 떨어지면
  - 4-1. 부품을 키우거나
  - 4-2. 후보를 늘리고반복한다.



<반례 기반 합성기>

입력: 합성 조건

출력: 합성한 최종 프로그램

합성 조건에서 최초 입출력 쌍을 꺼낸다.

- 1. <입출력 기반 합성기>에 최초 입출력 쌍을 던진다
- 2. 출력 결과가 합성 조건에 맞는지 Solver에 물어본다
  - 2-1. 맞으면 최종 정답
  - 2-2. 틀리면 Solver가 반례를 돌려준다.  
이 반례를 입출력 쌍에 더하고 반복한다.

# 경험: 부품을 빨리 키우는게 유리한 경향

component size	hd-09-d5	hd-13-d5	hd-14-d5	hd-15-d1
1	231.1	157.2	27.8	469.3
2	29178.6	81.1	2978.3	218284.1
3	106.1	94.8	24.5	105.7
4	531.4	0.7	2902.3	899.6
5	1.3	33.4	2.6	9.5

\* hacker's delight 문제, 짝수 분석 없는 양방향 합성, 부품 최대 크기는 한 번 만들어 고정하고 후보만 늘리는 방식, 편차는 찾아낸 정답 식의 차이와 생성되는 반례의 차이 등 때문

# 경험: 부품을 빨리 키우는게 유리한 경향

- 합성 자체의 성능에도 유리함
- 싹수 분석의 효율에도 유리함
  - 빈칸이 하나뿐인 후보라 해도 부품 수가 10배 차이나면 효과도 10배 차이
- 메모리가 넘치지 않는다면 어느 선까진 빨리 키워두는게 좋다
- 부품 키우기와 후보 키우기를 어느 시점에 하는게 좋은지 판단 필요 (더 많은 경험)

재활용으로 싹수 분석 효율 높이기

# 착안

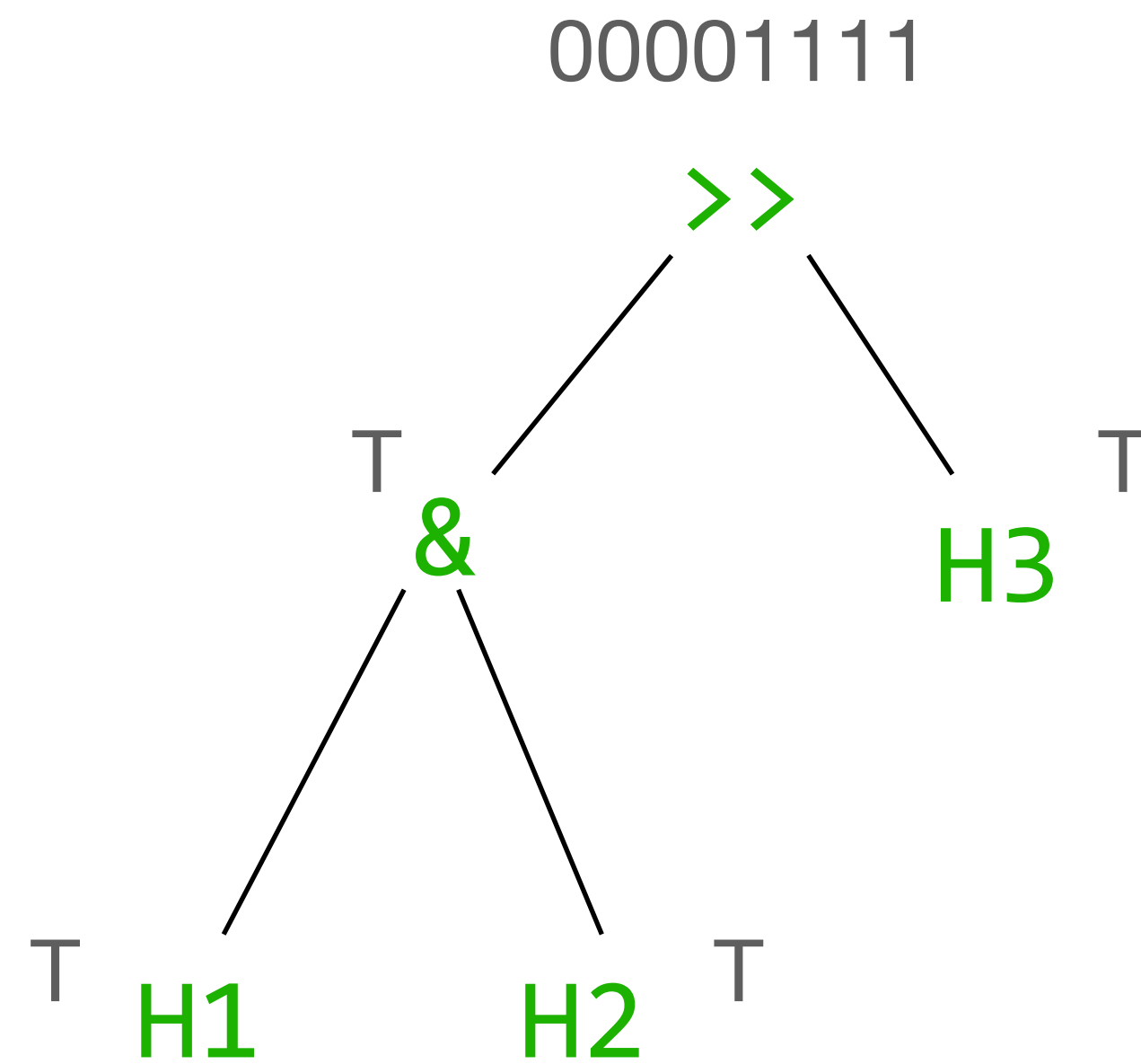
- 큰 차이가 없는 프로그램들에 대해 싹수 분석을 반복함
- 특히 빈칸만 더 채운 프로그램이라면?
  - 안전한 분석을 설계했다면 빈칸을 채운 프로그램은 채우기 전 프로그램보다 ‘구체적인’ 분석 결과가 나올 것
  - 후보 나열도, 부품 끼워보기도 빈칸 채우기로만 이루어짐
- 이전 분석 결과를 활용할 수 있지 않을까?

```
([]&[])>>[]  
([]&[])>>[1]  
([]&[x])>>[1]
```



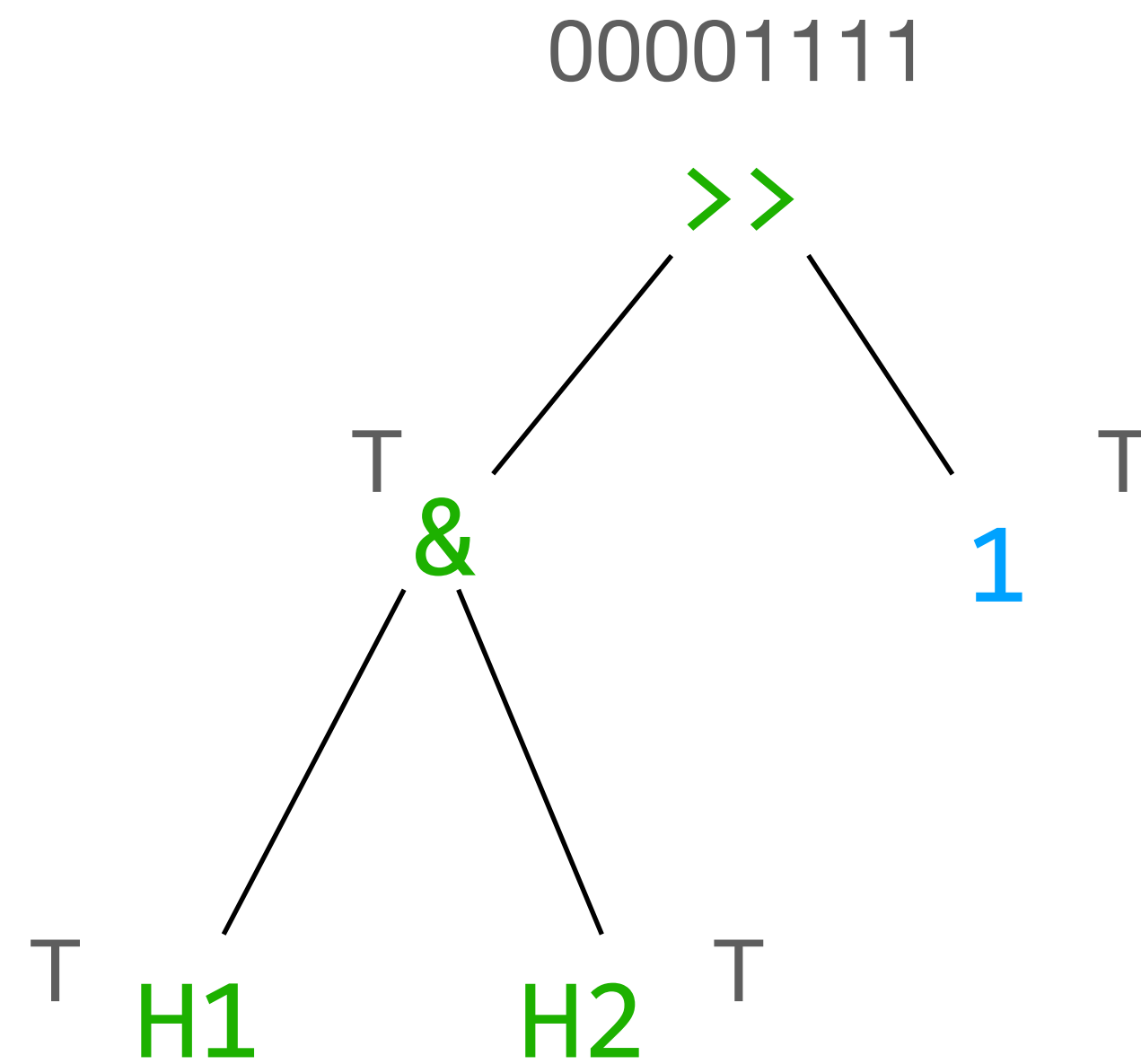
# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111
- 아직 정보가 없는 상태





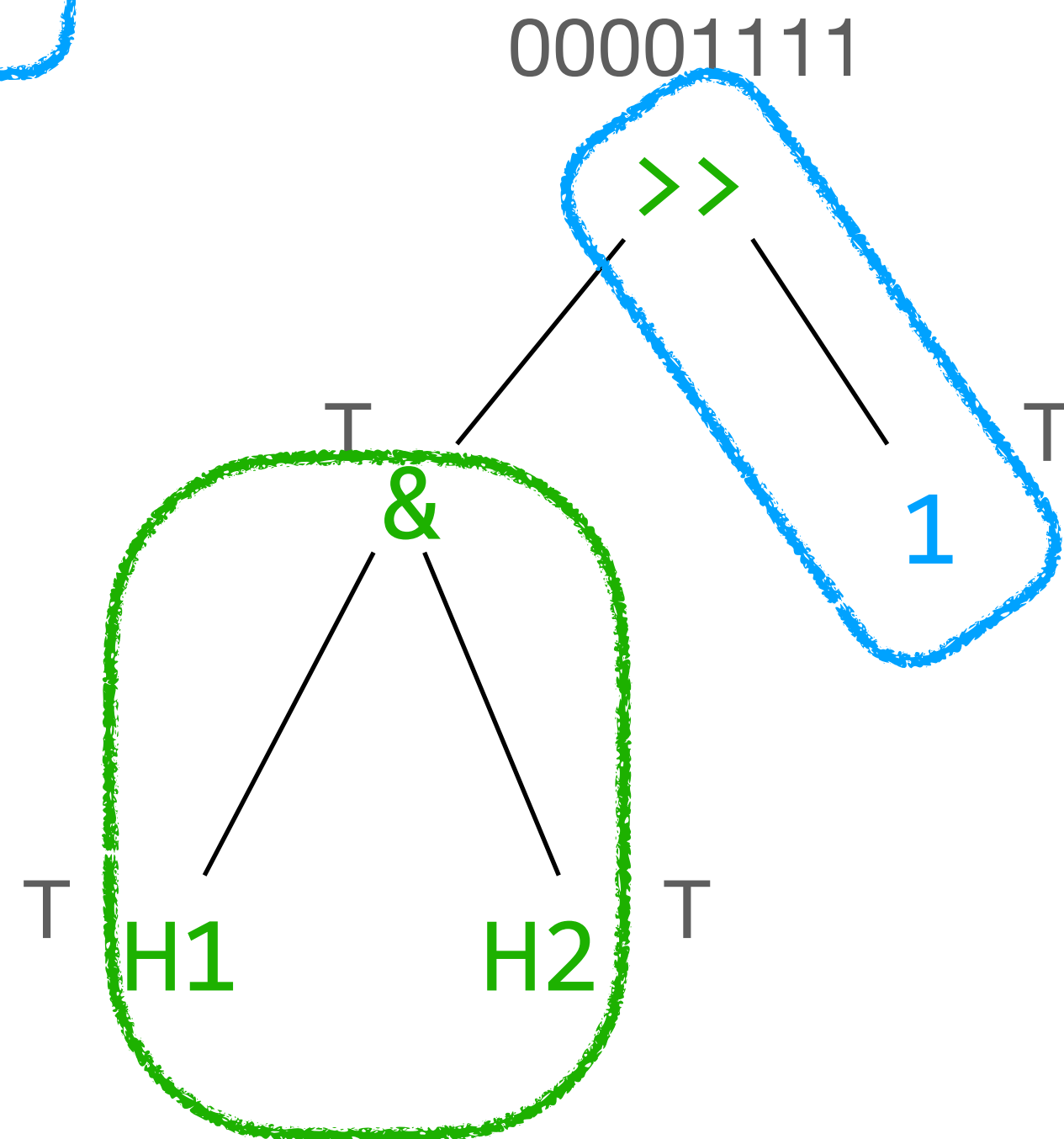
# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111
- 빈칸 H3 <- 부품 1



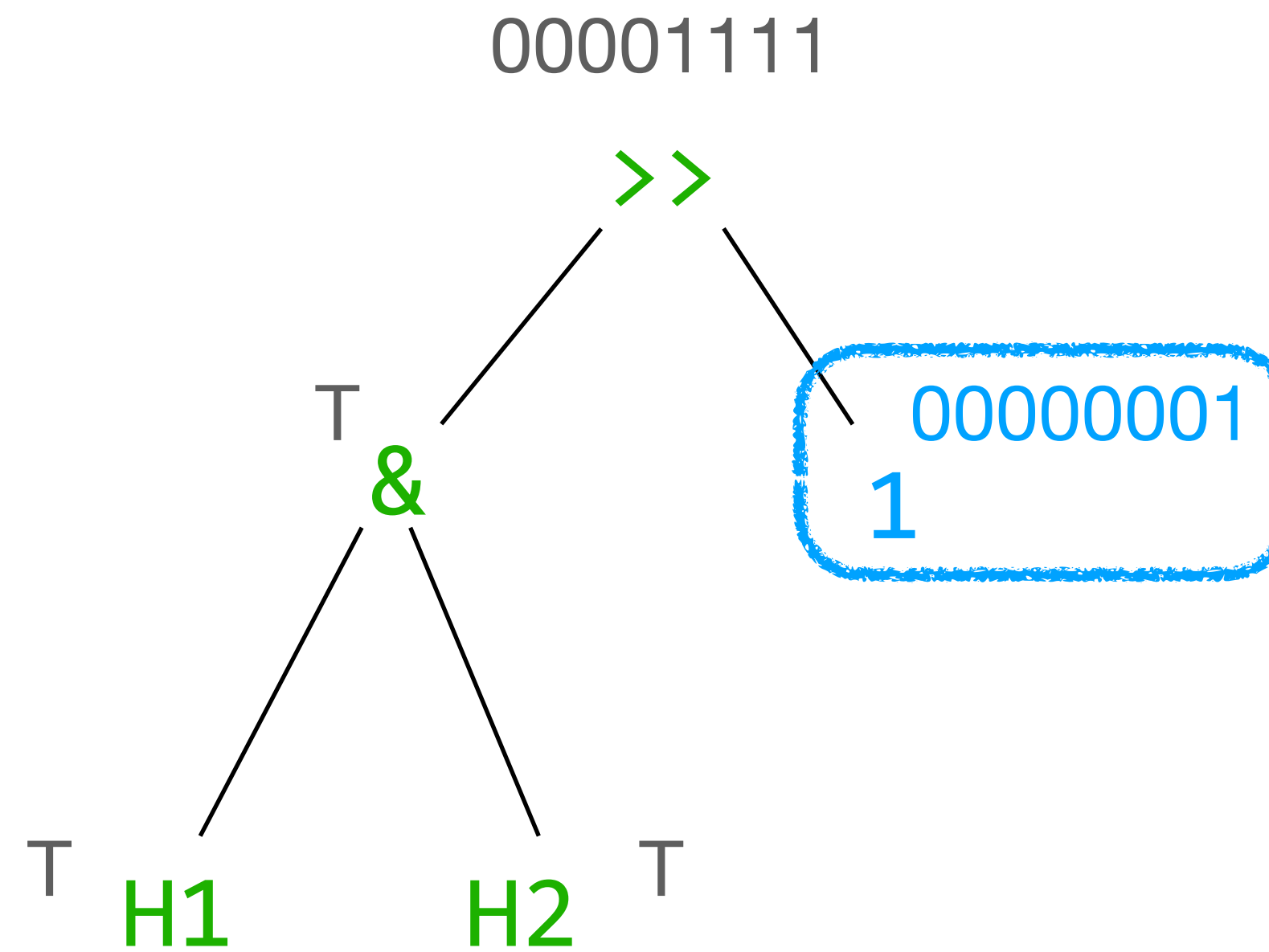
# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111
- 다시 정방향 분석을 해야하는 곳: 
- 재활용으로 시간 아낀 곳: 



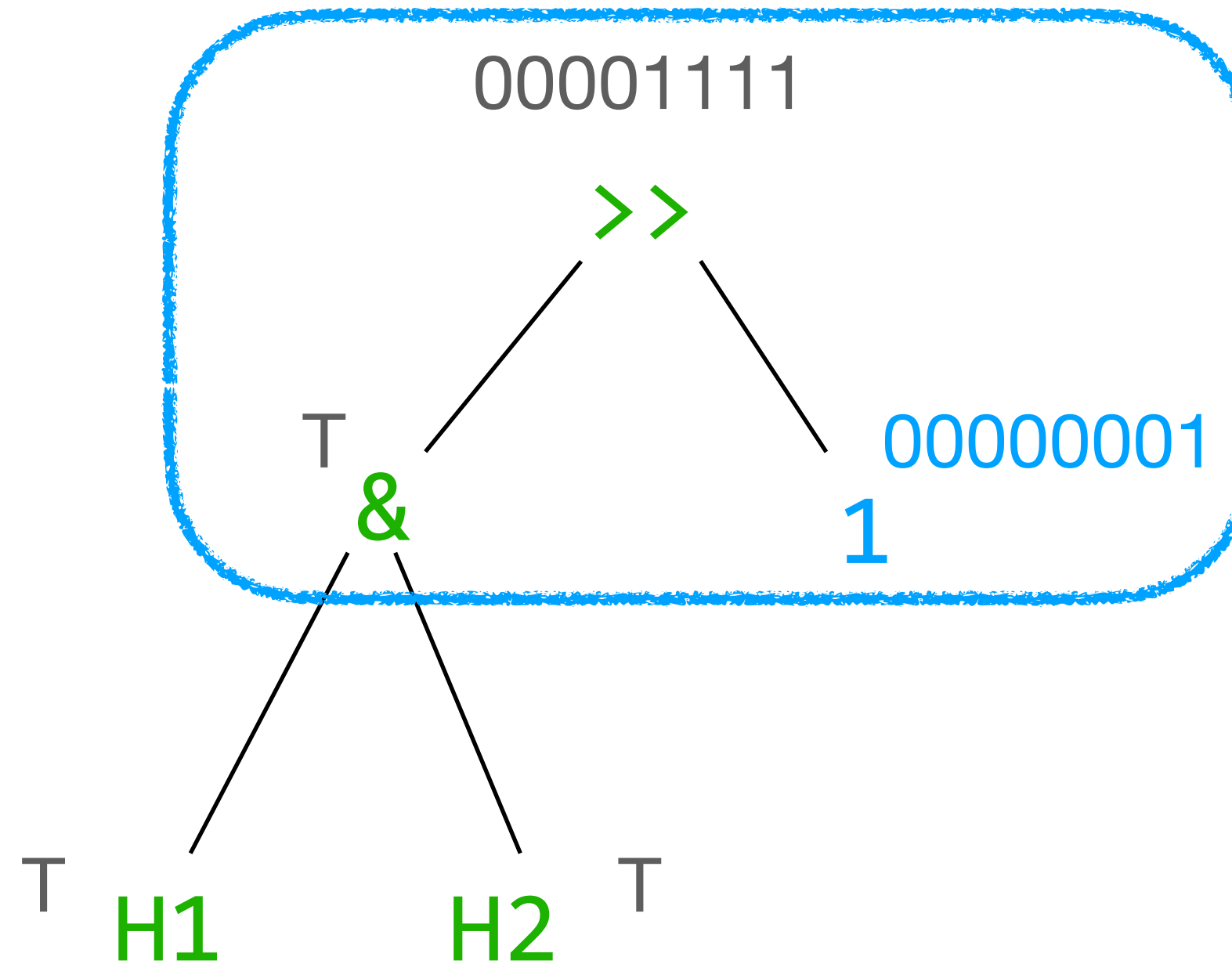
# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111
- 재활용 정방향 분석 결과



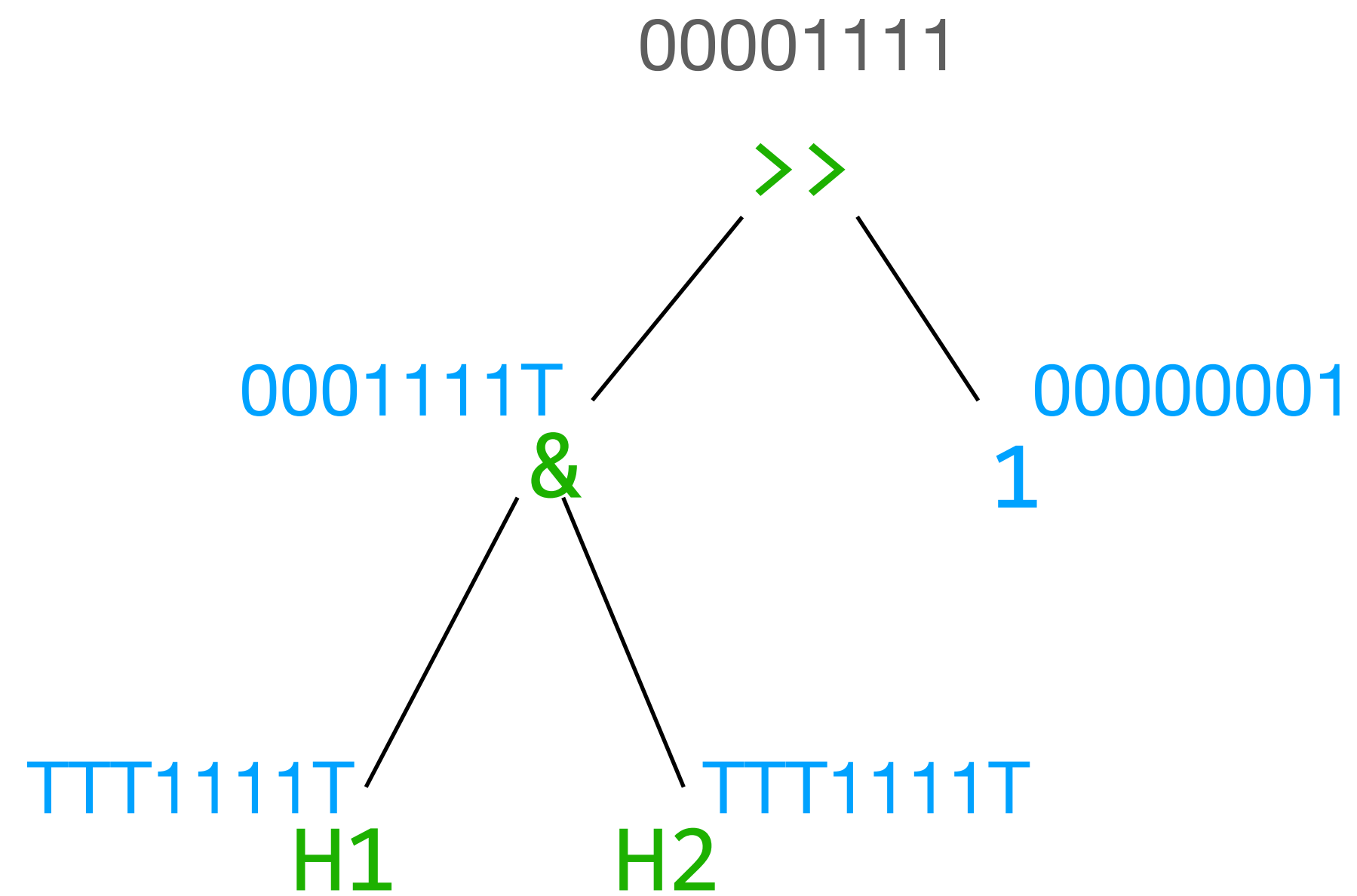
# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111
- 역방향 분석을 시작해야하는 지점:  
마지막 업데이트 된 곳을 포함하는  
가장 낮은 위치 = >>



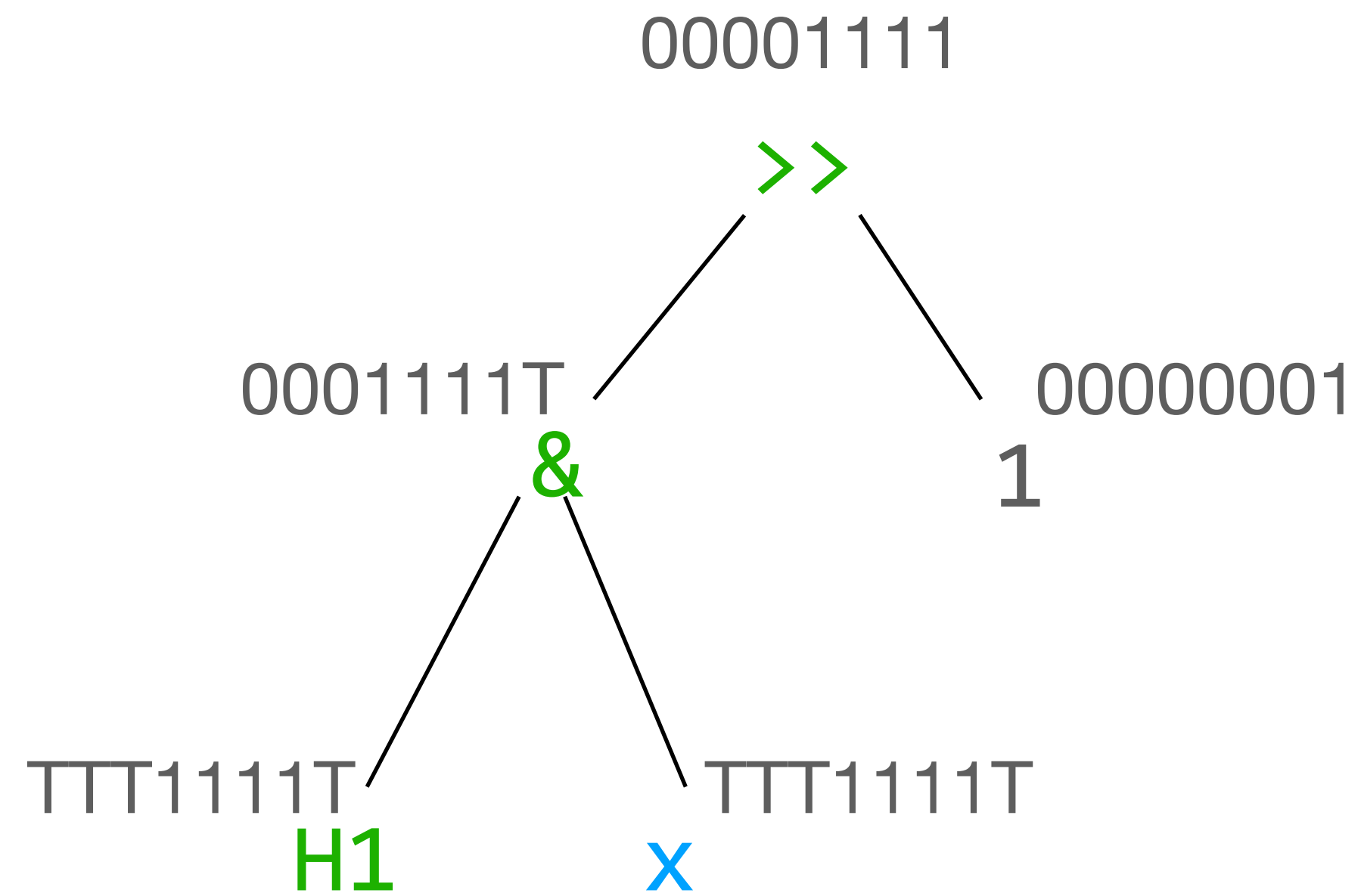
# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111
- 역방향 분석 완료  
아직 싹수 있음




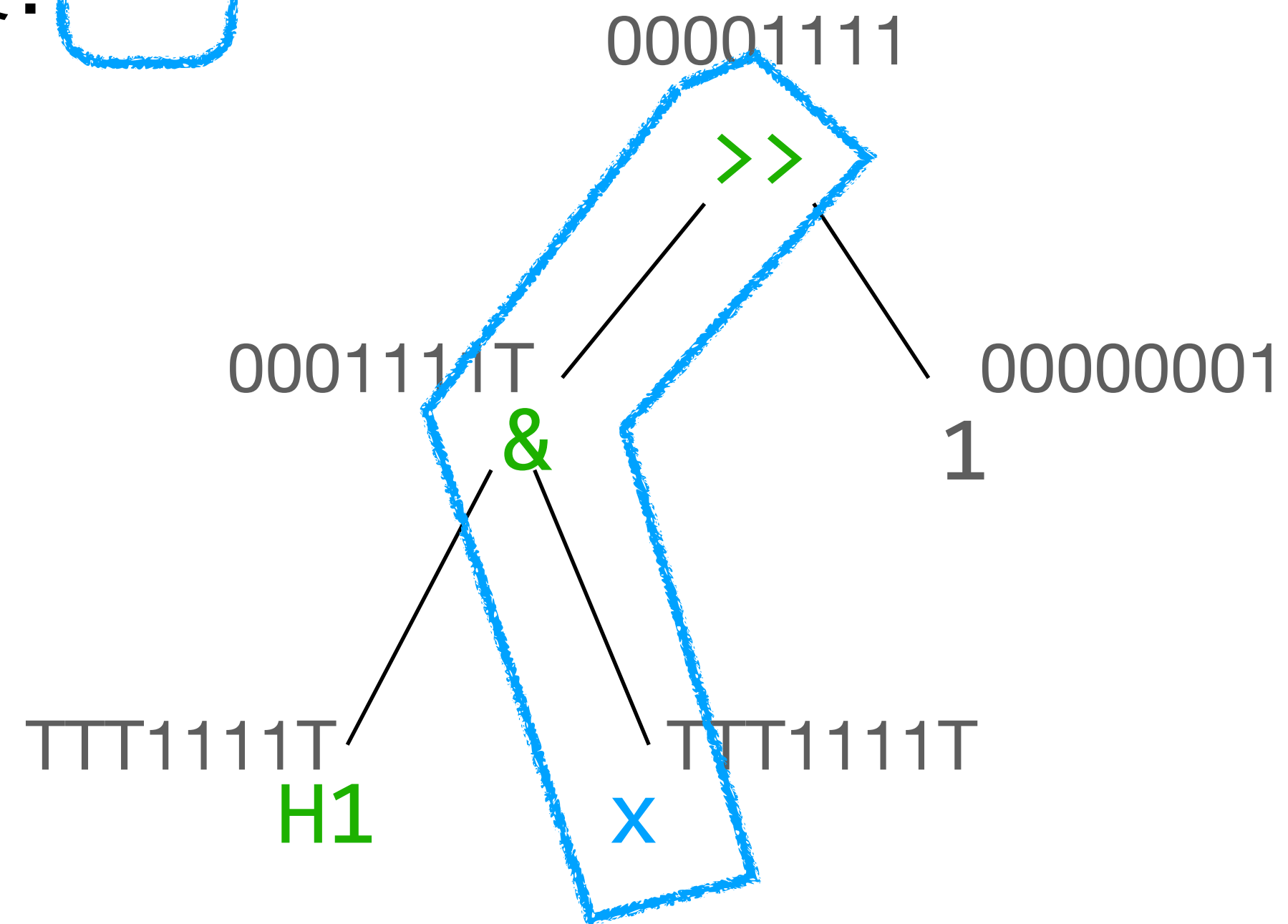
# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111
- 빈칸 H2 <- 부품 x



# 앞 결과 재활용하기

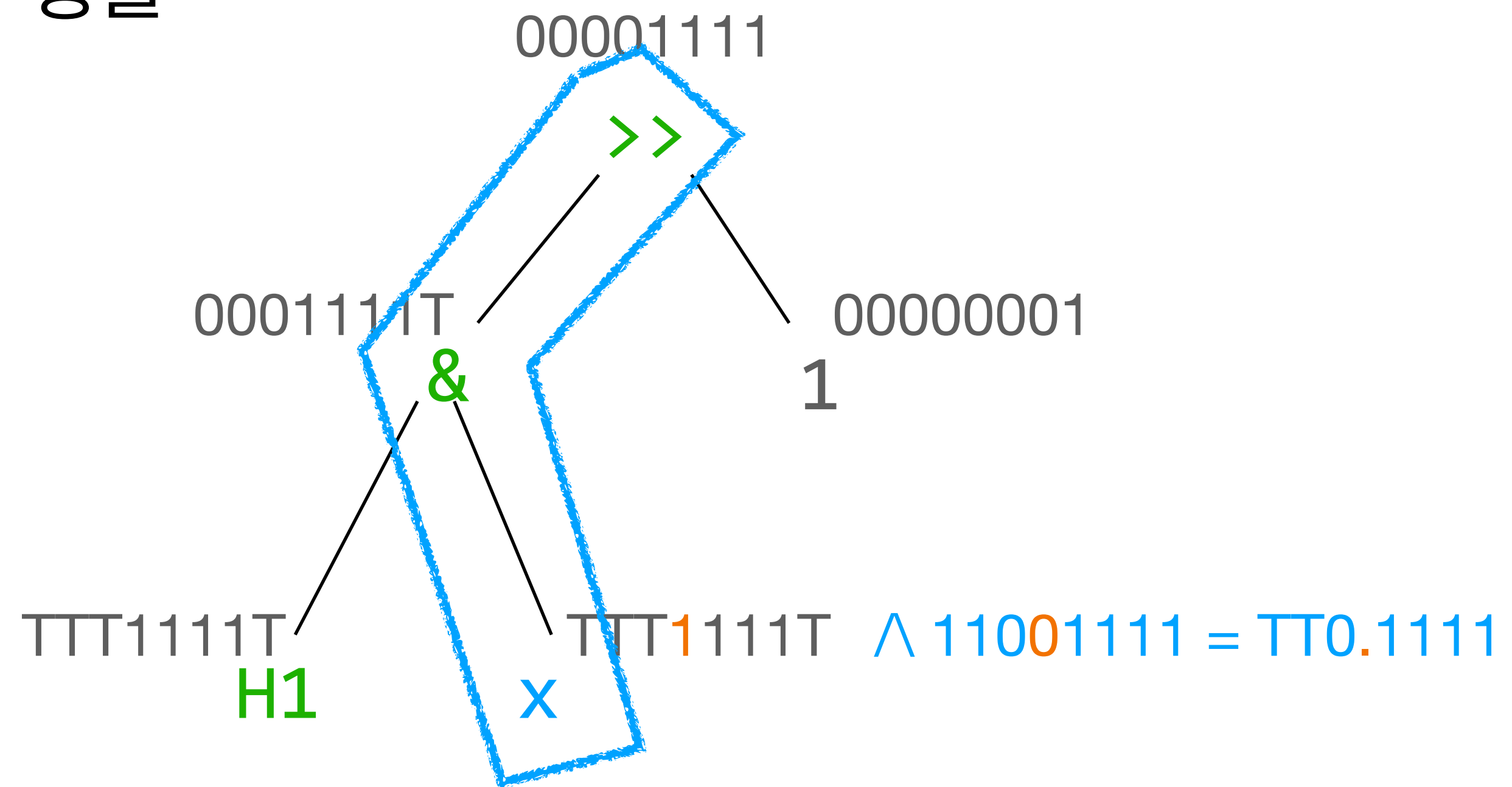
- 입력: b\_11001111 / 출력: b\_00001111
- 다시 정방향 분석을 해야하는 곳: 





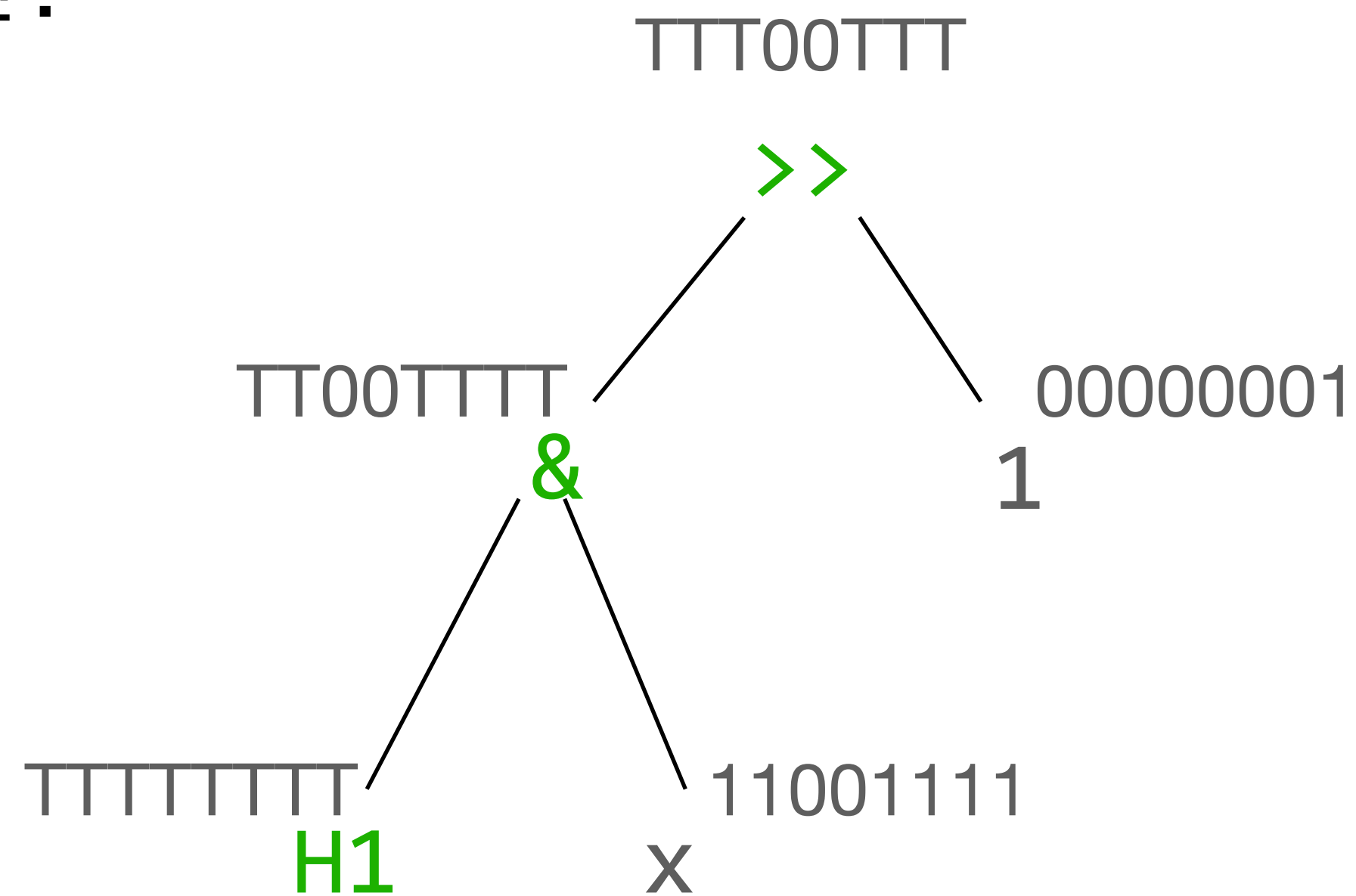
# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111
- x 정방향 분석 결과가 앞 결과와 충돌  
=> 실패 없음



# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111
- 무작정 다시 전체 분석을 했다면?
  - 전체 정방향 분석 후

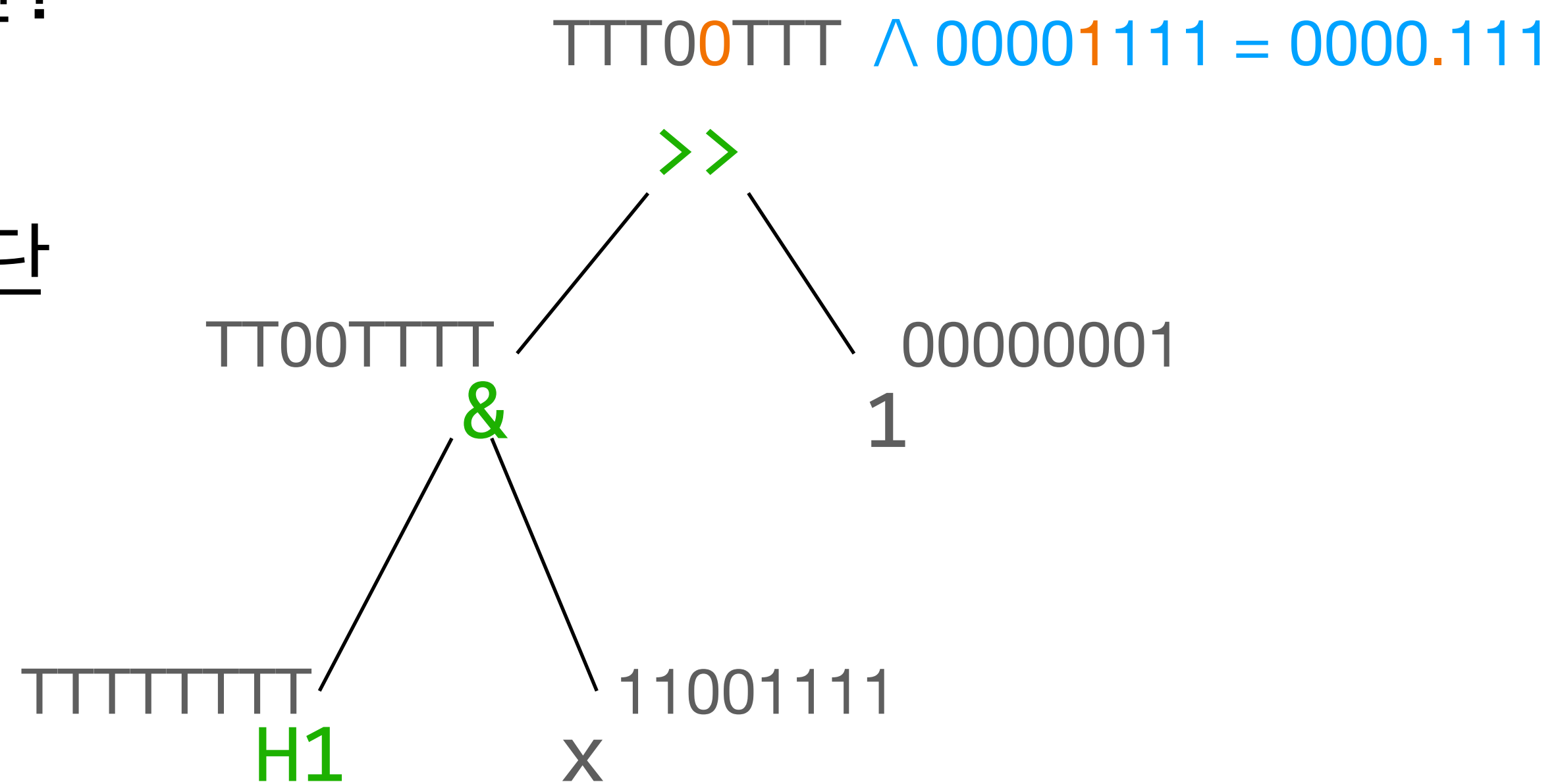


# 앞 결과 재활용하기

- 입력: b\_11001111 / 출력: b\_00001111

- 무작정 다시 전체 분석을 했다면?

- 전체 정방향 분석 후  
역방향 분석 중 실패 없음 판단



# 재활용 분석 방법

- 재활용 분석 입력: 빈칸에 어떤 식을 끼워넣은 프로그램  $P[E]$  와  $P[]$ 에 대한 분석 결과  $S$
- 분석 방법
  - 정방향:  $P[E]$  에서  $[]$  에 직접 영향을 받는 (= 나무 구조에서 부모-자식 관계에 있는) 노드만 새로 분석하고 나머지는  $S$ 에 매달린 값을 유지
  - 역방향: 정방향 분석에서 업데이트 된 가장 높은 위치를 포함하는 가장 낮은 노드부터 분석 시작

# 재활용 분석의 효과

problem	부품 크기	부품 갯수	분석없이	재활용 없이 분석		재활용 분석		
			총 시간	총 시간	분석 시간	총 시간	분석 시간	분석 시간 비율
hd-07-d5	1	4	0.11	3.93	3.81	2.42	2.30	60.5%
hd-08-d5	1	4	0.11	3.92	3.80	2.39	2.27	59.9%
hd-09-d0	1	7	1.84	22.64	21.20	10.34	8.93	42.1%
hd-09-d0	2	7	1.85	22.63	21.16	10.26	8.84	41.8%
hd-09-d1	1	36	281.12	1963.61	1686.43	925.23	653.15	38.7%
hd-09-d1	2	36	281.79	1965.85	1687.70	928.53	655.35	38.8%
hd-09-d1	3	36	4.01	8.28	5.47	8.32	5.50	100.6%
hd-09-d1	4	36	3.99	8.32	5.49	8.33	5.51	100.5%

\* hacker's delight 문제, 분석 포함 총 1초 이상 걸리고 합성에 성공한 문제만

# 재활용 분석의 효과

- 분석 시도할 후보가 많을 때 시간 절약 효과는 확실
- 메모리 효율 개선 필요
  - 후보 하나당 요약값 테이블 하나
  - 후보가 수십만개 이상 쌓이면 Out of Memory 발생

# 균형점을 찾아야

- 합성이 잘 되려면
  - 부품을 짝짝 키워야
- 분석을 효율적으로 하려면
  - 후보에 구멍이 많아야 -> 후보부터 키워야
- 희망: 아주 큰 프로그램을 합성할 때는 후보와 부품이 모두 커질 것