3. 예측파서

충북대학교

이 재 성

학습내용

- 하향식 파싱 방법
- 하향식 파서 설계 및 코드
- 왼쪽 순환 문법의 처리

파싱(PARSING)

파싱방법

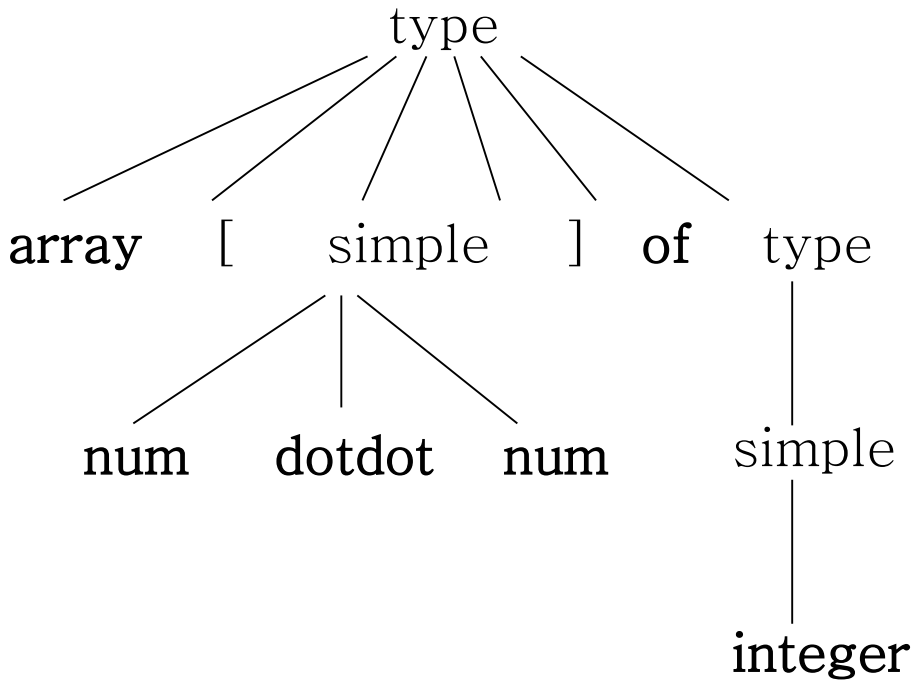
- 하향식(top down)
  - 파스트리를 루트에서 시작하여 아래로 구성해 가는 방식
  - 쓸만한 파서를 쉽게 만들 수 있는 방법
- 상향식(bottom up)
  - 파스트리를 잎노드에서 시작하여 위로 구성해 가는 방식
  - 더 넓은 범위의 문법과 번역계획을 처리할 수 있음

이것도 어차피 같은 가지 개수에서 가로로 보면 그렇게 변한다는 느낌임..
직접 만들어보면 앎.

하향식 파싱법 (예)

- 문법

type → simple
| ^ id
| array [simple] of type
simple → integer
| char
| num dotdot num



- 입력 토큰열

array [num dotdot num] of integer

결국 최종적으로 leaf nodes == 입력 토큰열이 되게 만들면 됨

leaf node만 모음

예측파싱법

- 예측 파싱법

재귀적-내림차순 파싱법

  - 순환적 내림차순 파싱법(Recursive-descent parsing)
  - 예측기호(lookahead)를 이용하여 파싱시에 백트랙킹을 하지 않는 방법
  - 입력을 처리하는 프로시저들의 실행 순서가 바로 파스트리가 됨
- 백트랙(backtrack)
  - 비단말에서 생성규칙을 선택할 경우, 주어진 입력에 맞지 않아 다른 생성규칙을 선택하기 위해 재시도를 하는 것
- First(α)
  - 비단말 기호 α로부터 생성된 하나 이상의 문자열의 첫번째 기호로 나오는 토큰들의 집합
  - 예: FIRST(simple) = {integer, char, num}
  - 예측기호가 First(α)에 속하면 α규칙이 적용

simple → **integer**
| **char**
| **num dotdot num**

예측파서 의사코드(pseudo code)-1

```
procedure match(t:token);
begin
        if lookahead = t then
                lookahead := nexttoken()
        else error
end;

procedure type();
begin
        if lookahead is one of {integer, char, num} then
                simple()
        else if lookahead = '^' then begin
                match('^'); match(id)
        end
        else if lookahead = array then begin
                match(array); match('[');simple();match(']');match(of);type()
        end
        else error
end;
```

시작: type 프로시저

입력렬:

array [num dotdot num] of integer

초기 lookahead=array

LHS

body

type → simple
| ^ id
| array [simple] of type

예측파서 의사코드(pseudo code)-2

```
procedure simple();
begin
        if lookahead = integer then
                match(integer)
        else if lookahead = char then
                match(char)
        else if lookahead = num then
                match(num); match(dotdot); match(num)
        end
        else error
end;
```

simple → **integer**
| **char**
| **num dotdot num**

* 이해를 돕기 위해 type, simple, nexttoken 프로시저에 ()를 포함시킴. 교재와 차이가 있음에 주의할 것

ε규칙의 사용

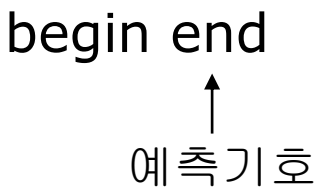
- 순환적 내림차순 파서에서 ε규칙 사용
  - 사용할 생성규칙이 하나도 없을 때 ε규칙 사용

- 예

  stmt → **begin** opt_stmts **end**

  opt_stmts → stmt_list | ε

  - opt_stmts 파싱시에 예측기호가 FIRST(stmt_list) 집합에 속하지 않는다면 ε규칙 사용
  - stmt가 begin end 인 경우 올바른 선택 (예측 기호가 end인 상태)



예측파서의 설계

- 프로시저의 2가지 일
  - 예측기호를 읽고 어떤 생성규칙을 사용할 것인지 결정
    - 예측기호가 FIRST(α)에 속하면
      - α를 오른쪽에 가진 생성규칙 선택
    - 예측기호에 대해 두개 이상의 생성규칙이 존재하면
      - 예측 파싱이 불가
    - 예측기호가 어떤 규칙의 FIRST(α)에도 속하지 않으면
      - 오른쪽에 ε가 있는 규칙 사용
  - 생성규칙의 오른쪽을 그대로 입력하듯 처리
    - 오른쪽의 비단말에 해당 프로시저를 수행
    - 예측기호와 일치하는 토큰은 다음 입력을 읽도록 함
    - 예측기호와 토큰이 일치하지 않으면 오류처리

왼쪽 순환(left recursion)

- 순환 내림차순 파서에서의 무한 반복
  - expr → expr + term

```
procedure expr();
begin
        if lookahead is one of {...} then
                expr();match('+');term()
        else
        :
end;
```

왼쪽 순환(left recursion)

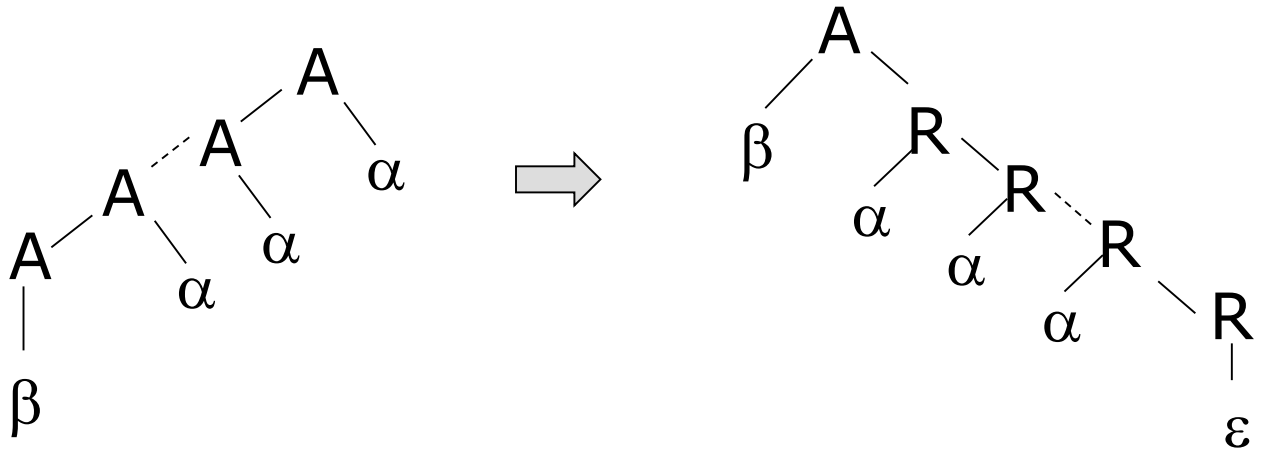
- 왼쪽 순환을 오른쪽 순환으로 변환
  - 왼쪽 순환적 생성규칙

    $A \rightarrow A\alpha \mid \beta$
  - 오른쪽 순환적 생성 규칙

    $A \rightarrow \beta R$

    $R \rightarrow \alpha R \mid \varepsilon$

★ 왼쪽 순환은 입력 토큰의 처리 없이 무한 반복되나, 오른쪽 순환은 입력 토큰을 하나씩 처리하여 마지막에 빈 토큰으로 처리하면 파싱이 가능해 짐



후위 표기 예측 파서

- 간단한 구문 중심 정의 (왼쪽 순환성)
  - 오른쪽 비단말의 번역결과를 원래 순서대로 연결하면서 약간의 문자열을 추가하여(또는 추가없이) 생성규칙의 왼쪽에 있는 비단말을 정의

  - 왼쪽 순환성 때문에 예측 파싱에 부적절

$expr \rightarrow expr_1 + term \quad \{ print('+') \}$

$expr \rightarrow expr_1 - term \quad \{ print('-') \}$

$expr \rightarrow term$

$term \rightarrow 0 \quad \{ print('0') \}$

$term \rightarrow 1 \quad \{ print('1') \}$

...

$term \rightarrow 9 \quad \{ print('9') \}$

왼쪽 순환성 제거 문법(잘못된 예)

- 수정 문법

  expr → term rest

  rest → + expr | - expr | ε

  term → 0 | 1 | ... | 9

- 문제점 예
  - rest → - expr 에 대해 생성규칙이 불명확

    1) rest → - {print('-')} expr

    2) rest → - expr {print('-')}
  - 1번 규칙의 적용 예

    9-5 => 9-5 (정답: 95-)
  - 2번 규칙의 적용 예

    9-5+ 2 => 952+ - (정답: 95-2+)

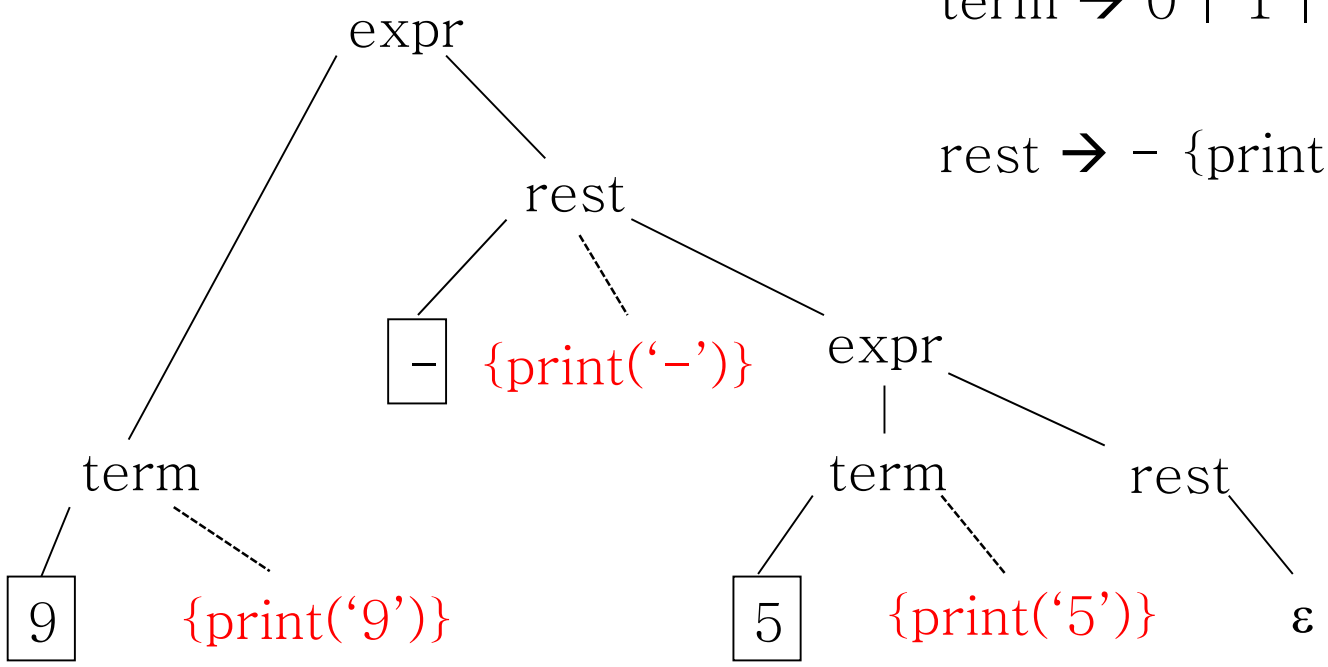
잘못된 문법 적용예(규칙1)

expr → term rest

rest → + expr | - expr | ε

term → 0 | 1 | ... | 9

rest → - {print('-')} expr



잘못된 문법 적용예(규칙2)

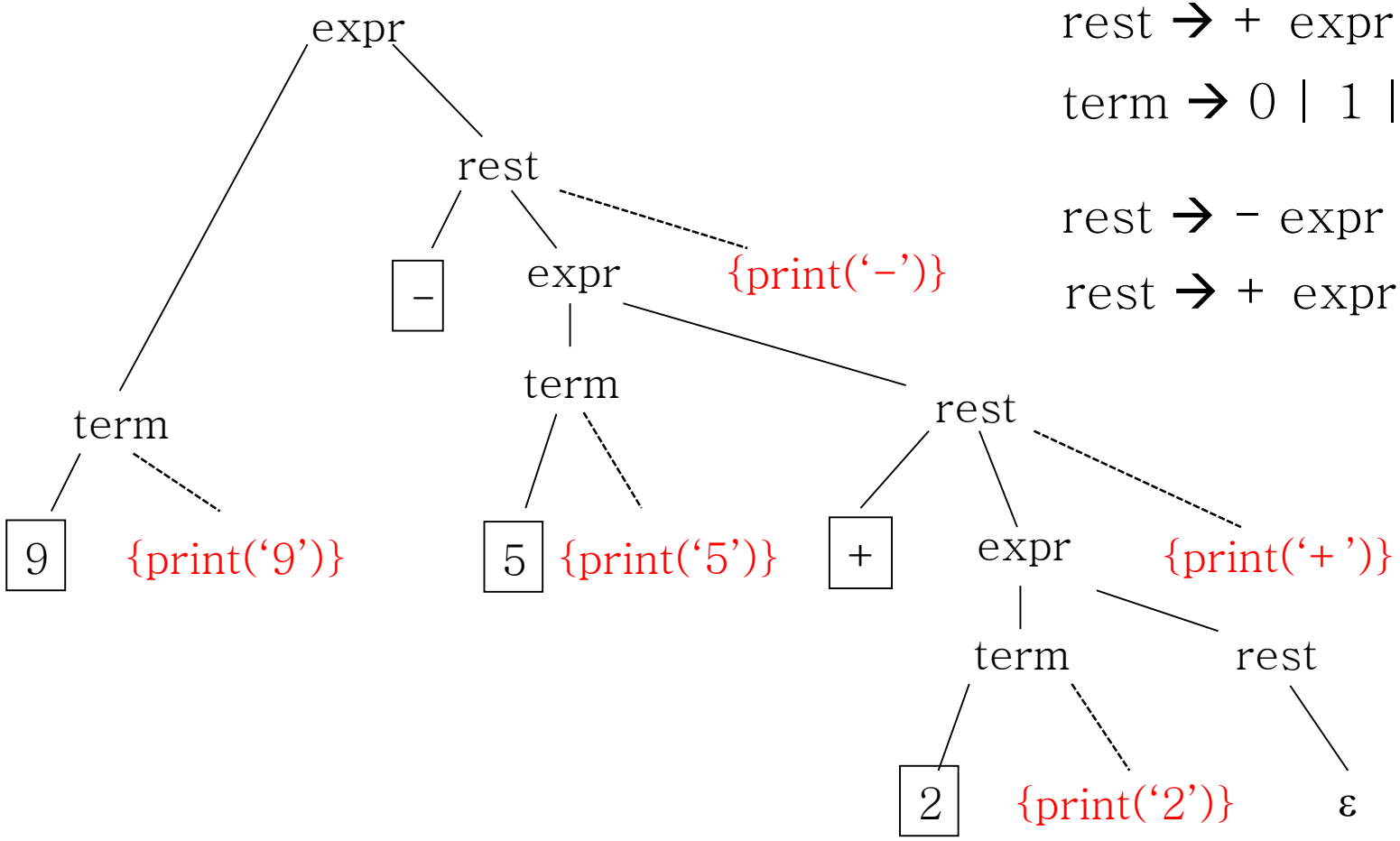
expr → term rest

rest → + expr | - expr | ε

term → 0 | 1 | ... | 9

rest → - expr {print('-')}

rest → + expr {print('+')}



올바른 번역 계획(왼쪽순환성 제거)

- 좌순환 제거 규칙 적용
  - A → Aα | Aβ | γ 의 경우 다음으로 변경

    A → γR

    R → αR | βR | ε
- 변환된 규칙

  expr → term rest

  rest → + term { print('+') } rest | - term { print('-') } rest | ε

  term → 0 { print('0') }

  term → 1 { print('1') }

  …

  term → 9 { print('9') }

올바른 번역계획의 수행예

expr → term rest

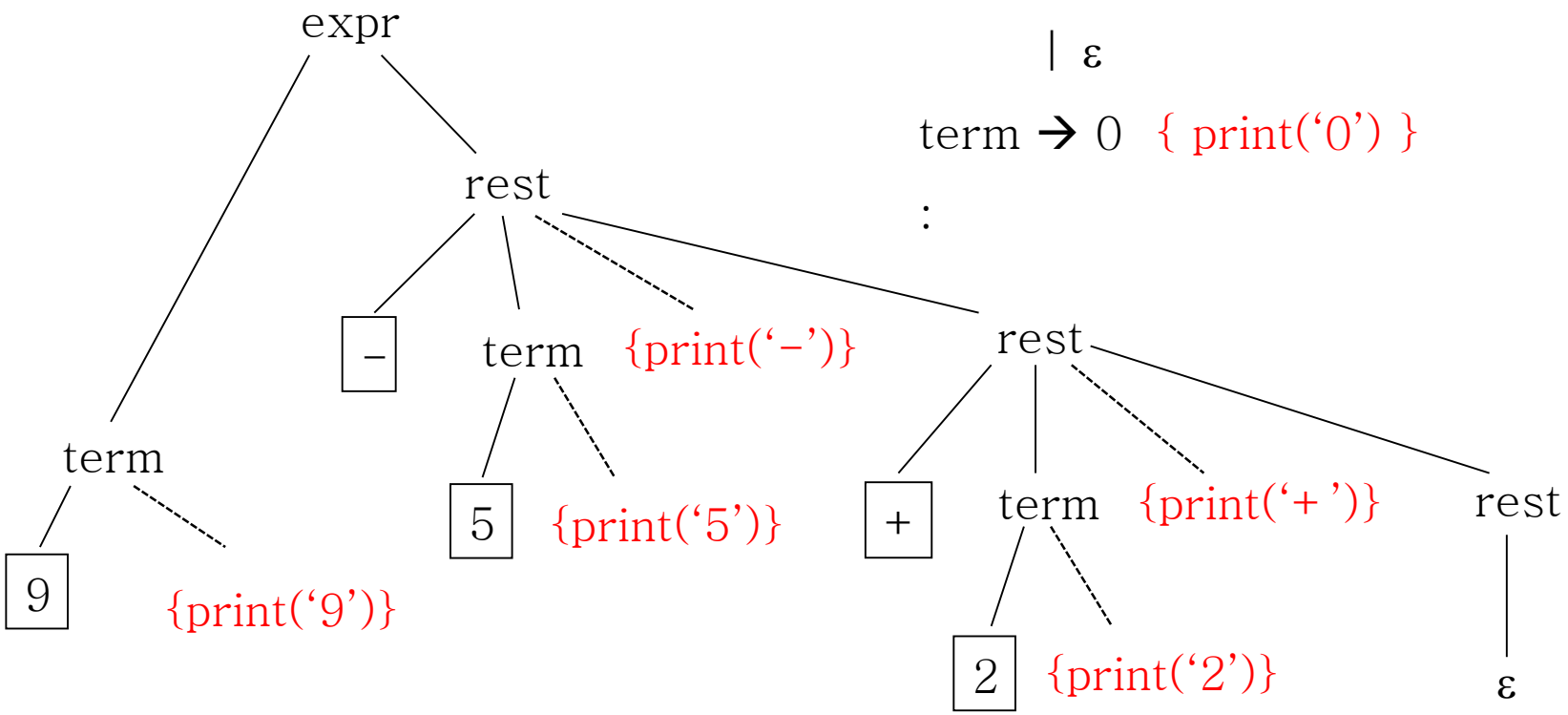
rest → + term { print('+') } rest

| - term { print('-') } rest

| ε

term → 0 { print('0') }

:



번역계획에 대한 코드

```
expr() {
        term(); rest();
}
rest() {
        if(lookahead == '+') {
                match('+'); term(); putchar('+'); rest();
        }
        else if (lookahead == '-') {
                match('-'); term(); putchar('-'); rest();
        }
        else ;
}
term() {
        if (isdigit(lookahead)) {
                putchar(lookahead); match(lookahead);
        }
        else error();}
```

꼬리순환

코드도 결국 작대기 이어진 것 만으로 이루어짐... 잘 보셈

꼬리순환의 해결-1(반복문 사용)

```
rest() {
L:      if(lookahead == '+') {
                match('+'); term(); putchar('+'); goto L;
        }
        else if (lookahead == '-') {
                match('-'); term(); putchar('-'); goto L;
        }
        else ;
}
```

꼬리순환의 해결-2(반복문 사용)

```
expr()
{
        term();
        while(1)
                if(lookahead == '+') {
                        match('+'); term(); putchar('+');
                }
                else if (lookahead == '-') {
                        match('-'); term(); putchar('-');
                }
                else break;
}
```

참고 문헌

- [1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, “Compilers – Principles, Techniques, and Tools,” Bell Telephone Laboratories, Incorporated, 1986.
- [2] 오세만, “컴파일러 입문”, 정익사, 2004.