Chapter 4
문법

- 어휘 구조 (렉시컬 구조; Lexical Structure)
- 신택스 구조 (Syntactic Structure)
- BNF, EBNF, 신택스 다이어그램 (Syntax Diagrams)
- 파스 트리 (Parse Tree), 신택스 트리 (Syntax Tree), 그리고 모호성 (Ambiguity)
- 파싱 기술과 도구
- 어휘 (Lexics) 대 신택스(Syntax) 대 구문 (Semantics)

소개 (Introduction)

- 형식적인 언어 구조 (촘스키) 🡐 중요
  - 정규 언어 (Regular)
  - 문맥 자유 또는 컨텍스트 프리 (Context-Free)
  - 문맥 민감 또는 컨텍스트 센서티브(Context-Sensitive)
  - 무제한 (Unrestricted)
- 프로그래밍 언어의 신택스 구조 🡐 중요
  - 렉시컬 구조 (단어들의 구조)
    - 정규식으로 나타내어질 수 있음
  - 신택스 구조 (문장들의 구조)
    - 컨텍스트 프리 문법 (CFG)으로 나타내어질 수 있음
    - BNF (Backus-Naur Form) 은 CFG를 나타내는 방법

렉시컬 구조

- 렉시컬 구조
  - 토큰의 구조
  - 토큰 : 프로그래밍 언어의 단어들
    - 토큰은 단어들의 집합
- 정규식
  - 토큰의 패턴을 기술하기 위한 것
  - 텍스트 검색에 많이 쓰임

토큰

- 토큰의 종류들 (C 언어 예)
  - 키워드 `if`
  - 리터럴 (상수) `"if"`
  - 특별 심볼(special symbols) `;`
  - 식별자 (identifiers) `IF`
  - 커멘츠 (대부분 공백으로 취급됨) `/*if*/`
- 식별자의 최대 길이는 언어보다는 구현에 의해 결정된다.

렉시컬 구조 이슈들

- 프로그램 레이아웃
  - 자유 포맷 (free format)
  - 레이아웃의 규칙들 (오프사이드 규칙 – offside rules)
  - 고정 포맷 (fixed format)
- 흰 공백이 무시되는가 안되는가?

```
DO 99 I = 1,10
DO 99 I = 1.10
```

- 키워드가 예약되었는가 아닌가?
  - 어떤 언어들의 키워드는 예약되지 않음
  - 포트란(FORTRAN)이 그러한 언어의 예임

```
IF(IF.LT.0) IF = IF + 1
ELSE IF = IF + 2
```

정규식 ← 중요

- 패턴을 기술하는 언어
  - 어떤 토큰들의 구조는 매우 복잡하므로, 이러한 구조를 표현하기 위한 표현방법이 필요함
  - 정규식은 토큰의 패턴들을 기술하기 위해 사용됨
- 정규식의 예
  - 숫자들의 시퀀스

    `[0-9]+`
  - 부호없는 부동소수점 리터럴

    `[0-9]+(\.[0-9]+)?`

스캐너 (Scanner) ← 중요

- 토큰을 인식하는 프로그램
  - 스캐너: 프로그램 소스를 스캔해서 토큰들을 인식해서 시퀀스로 반환하는 프로그램
  - 스캐너 구축 도구: 정규식으로 쓰여진 스캐너의 스펙을 입력받아 스캐너를 구축하는 프로그램
- 스캐너의 예 (그림 4.1)
  - 토큰의 종류: **`TokenType`**
  - 스캐너: **`TokenType getToken(void)`**
  - 스캐너의 드라이버: **`main()`**

컨텍스트 프리 문법

- BNF 예 (간단한 영어 문장)
  (1)문장 → 명사구 동사구 .
  (2)명사구 → 관사 명사
  (3)관사 → a | the
  (4)명사 → girl | dog
  (5)동사구 → 동사 명사구
  (6)동사 → sees | pets
- 용어
  - 문법: 다시쓰기 규칙들의 집합 (a set of rewriting rules)
    - (메타심볼(metasymbols) : →, |)
  - 논터미널(nonterminals): 화살표의 왼쪽
    - (LHS'es of the arrow (예를 들어 관사))
  - 터미널(terminals): 토큰들 (예를 들어 .)
  - 시작 심볼(the start symbol): 미리 정의된 논터미널, 첫번째 규칙의 LHS (문장)

CFG 는 "언어"를 생성함

- 언어
  - 언어: 문장들의 집합
  - 문장: 터미널들의 시퀀스
- 유도(Derivation)
  - 시작 심볼로부터 다시쓰기들의 시퀀스
  - CFG에 의해 기술되는 언어는 유도에 의해 정의될 수 있음

- 유도의 예

문장
⇒ **명사구 동사구 .**
⇒ **관사 명사 동사구 .**
⇒ **the 명사 동사구 .**
⇒ **the girl 동사구 .**
⇒ **the girl 동사 명사구 .**
⇒ **the girl sees 명사구 .**
⇒ **the girl sees 관사 명사 .**
⇒ **the girl sees a 명사 .**
⇒ **the girl sees a dog .**

CFG는 재귀적임

- 단순한 반복

  number → ***number digit***
  | ***digit***

  digit → 0 | 1 | 2 | 3 | 4
  | 5 | 6 | 7 | 8 | 9

- 유도의 예

  number ⇒ ***number digit***
  ⇒ ***number digit digit***
  ⇒ ***digit digit digit***
  ⇒ **2** ***digit digit***
  ⇒ **23** ***digit***
  ⇒ **234**

- 구조의 반복

  expr → ***expr*** **+** ***expr***
  | ***expr*** ***** ***expr***
  | **(** ***expr*** **)** | **NUMBER**

  NUMBER = [0-9]+

- 유도의 예

  expr ⇒ ***expr*** ***** ***expr***
  ⇒ **(** ***expr*** **)*** ***expr***
  ⇒ **(** ***expr*** **+** ***expr*** **) *** ***expr***
  ⇒ **… ⇒ (2 + 3) * 4**

렉시컬 구조 또한 CFG에
의해 표현될 수 있음

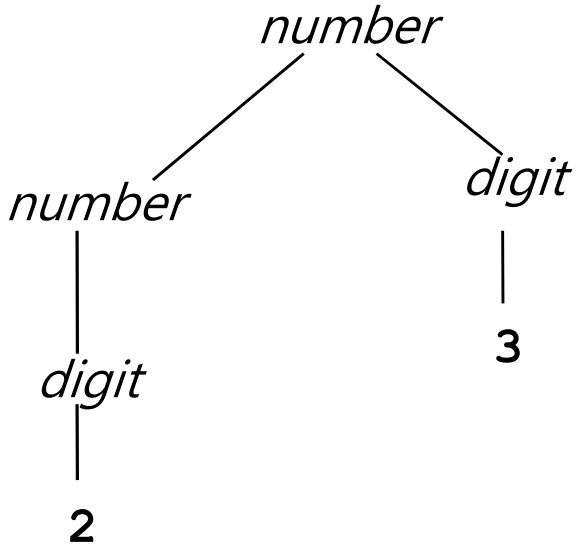
파스 트리(Parse Tree)

- 한 가지 관찰
  - 동일한 문장을 많은 다른 유도 방법으로 기술할 수 있음

    number ⇒ ***number digit*** ⇒ ***digit digit*** ⇒ ***digit*** 3 ⇒ 23

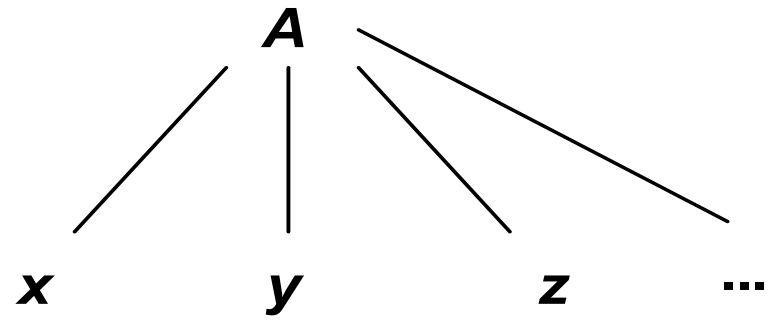
    number ⇒ ***number digit*** ⇒ ***digit digit*** ⇒ 2 ***digit*** ⇒ 23

- 파스 트리
  - 유도의 그래픽 표현



파스 트리에 대해서...

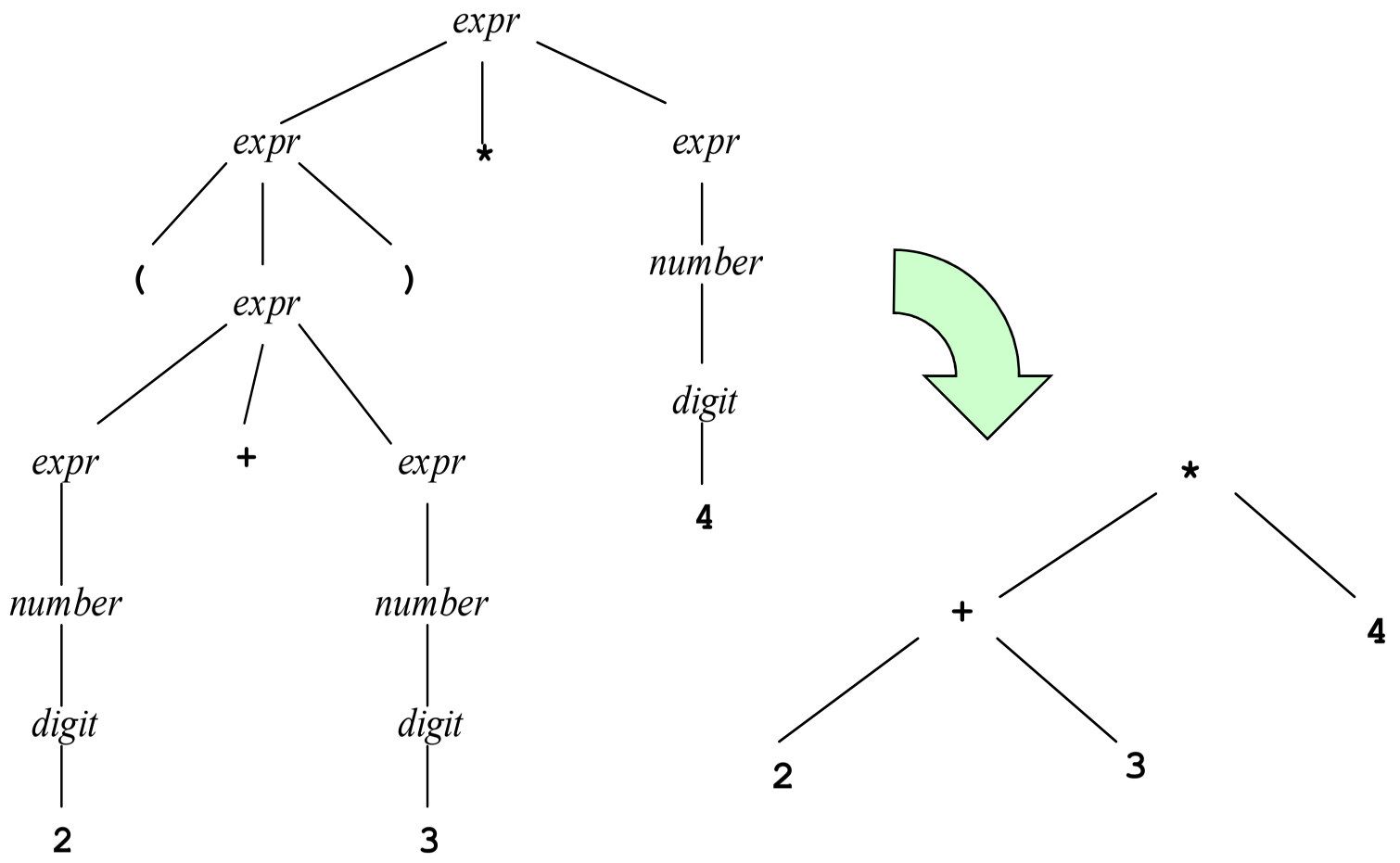
- 파스 트리 표현
  - 내부 노드는 논터미널
  - 리프 노드는 터미널
  - 루트 노드는 시작 심볼
  - 문법 규칙은 하나의 레벨을 가진 파스 트리를 구성
  - 예를 들어 A → xyz...



추상 신택스 트리 (Abstract Syntax Tree)

- 기본적인 관찰
  - 번역을 위해서 파스 트리 내의 정보가 다 필요한 것은 아님
- 추상 신택스 트리 (AST)
  - 요약된 파스 트리
  - 논터미널 노드는 없이지고, 일부 터미널 노드가 내부 노드가 됨

파스 트리와 AST의 예


expr
expr
*
expr
(
expr
)
number
expr
+
expr
digit
4
number
number
digit
digit
2
3
*
+
4
2
3

모호함 (Ambiguity)

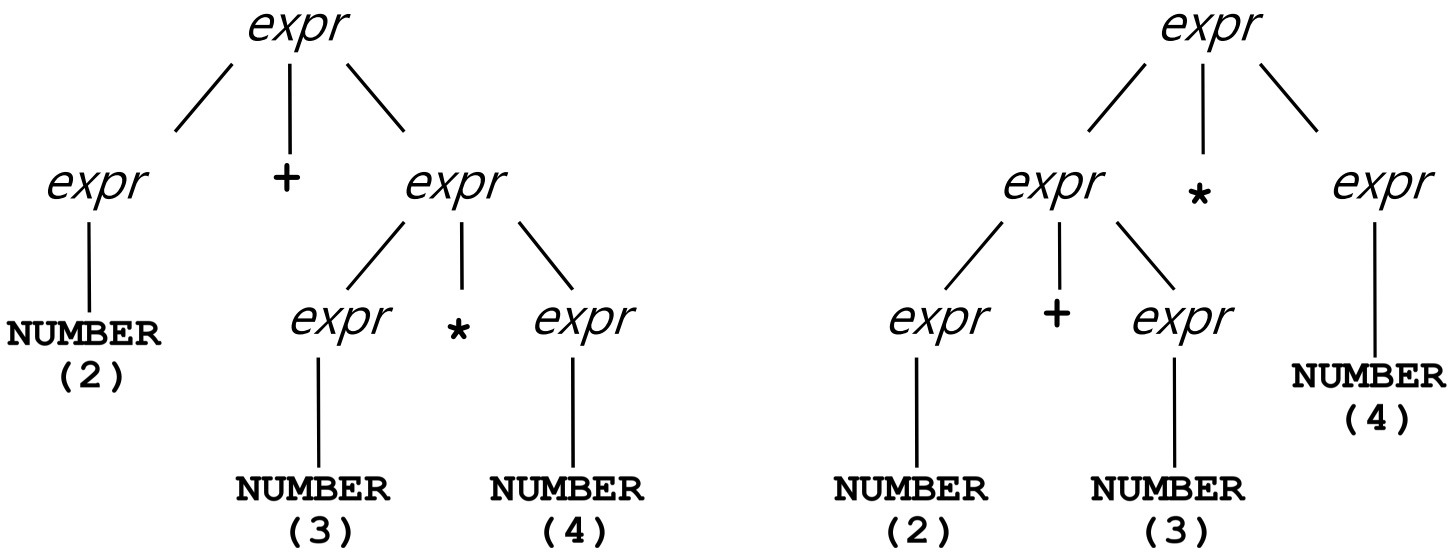
- 문법의 모호함
  - 만일 특정 문장이 주어진 문법에 의해 두 개 이상의 파스 트리 또는 AST를 가진다면, 그 문법은 모호함
  - 모호함은 어떤 문장들을 위한 구조들은 유일하게 하나만 있지 않다는 것
- 모호함을 다루는 방법
  - 어떤 파스 트리가 맞는지를 결정하기 위해 의미론적으로 검증
  - 올바른 선택을 위해 때로는 문법이 다시 쓰여짐

모호함의 예

- 문법:

  expr → ***expr*** **+** ***expr*** | ***expr*** ***** ***expr*** | **(** ***expr*** **)** | **NUMBER**

- 문장: **2 + 3 * 4**
- 파스 트리:

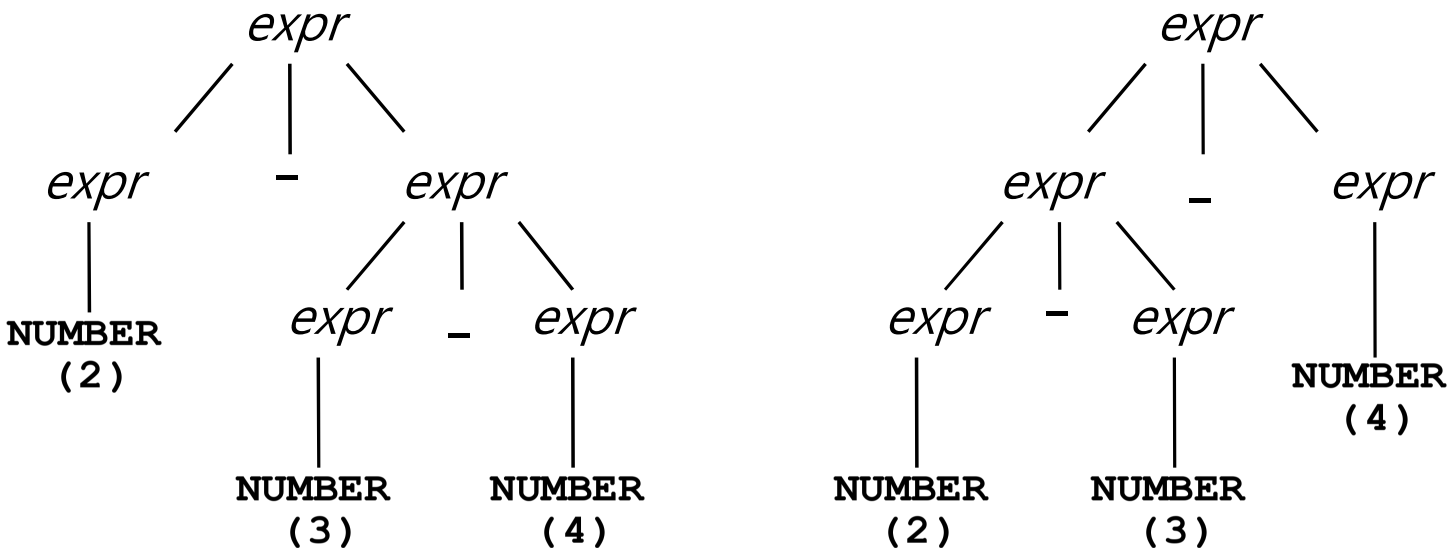


모호함의 다른 예

- 문법:

  expr → ***expr*** **+** ***expr*** | ***expr*** **-** ***expr*** | **(** ***expr*** **)** | **NUMBER**

- 문장: **2 - 3 - 4**
- 파스 트리:

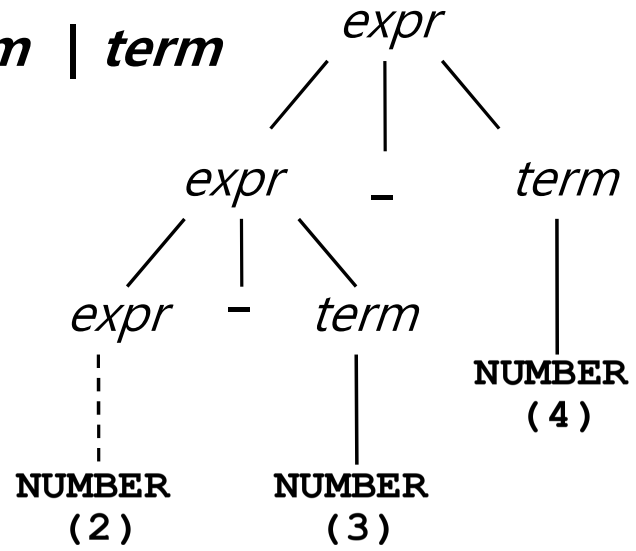


결합성(Associativity) 추가

- 결합성이 추가되어 논터미널의 재귀를 조절할 수 있음
- 좌 결합성을 주는 경우

expr → ***expr*** **+** ***term*** | ***expr*** **-** ***term*** | ***term***

term → **(** ***expr*** **)** | **NUMBER**



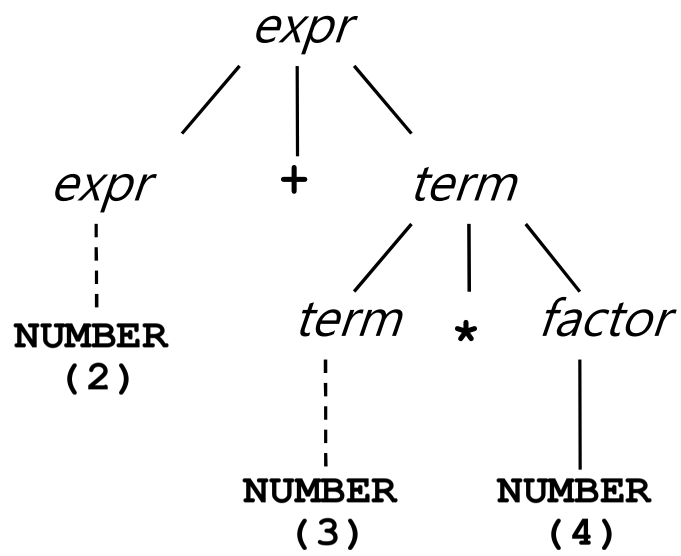
우선 순위(Precedence) 부과

- 비슷한 방법으로 연산자의 우선 순위를 부여할 수 있음 (우선순위 단계; "precedence cascade")
- * 를 + 보다 우선함

expr → ***expr*** **+** ***term*** | ***term***

term → ***term*** ***** ***factor*** | ***factor***

factor → **(** ***expr*** **)** | **NUMBER**



확장된 BNF (EBNF)

- 메타심볼들 추가됨
  - [ ] : 옵셔널한 구조
  - { } : 반복적인 구조
- 표현력
  - 메타 심볼이 추가되어도, EBNF의 표현력은 BNF와 동일함

EBNF의 예

- 반복
  - BNF: ***expr*** → ***expr*** `+` ***term*** | ***term***
  - EBNF: ***expr*** → ***term*** { `+` ***term*** }
- 옵션
  - BNF: ***if-stmt*** → `if (` ***expr*** `)` ***stmt***
    | `if (` ***expr*** `)` ***stmt*** `else` ***stmt***
  - EBNF: ***if-stmt*** → `if (` ***expr*** `)` ***stmt*** [`else` ***stmt***]

EBNF 사용에 관한 몇 가지 점

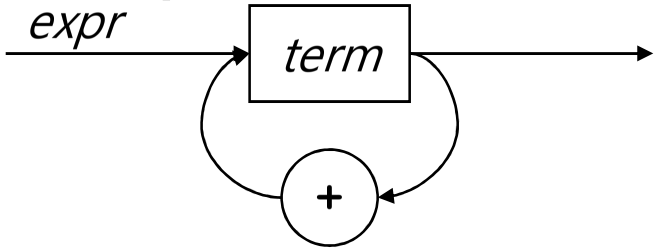
- 왼쪽으로 반복되는 규칙에 대해서만, {...}를 사용함:
  expr* → *expr* + *term* | *term
  는 다음과 같이 바뀜
  ***expr* → *term* [+ *expr*]**
- 규칙을 {...}로 시작하지 않음:
  ***expr* → *term* {+ *term* }라고 씀**
  ***expr* → {*term* +} *term* 는 틀렸음**
- [...]는 어디서든 쓰일 수 있지만, 다음은 조심:
  expr* → *expr* + *term* | *term* | - *term
  은 다음과 같이 쓰여져야 함
  ***expr* → [-] *term* {+ *term* }**

신택스 다이어그램

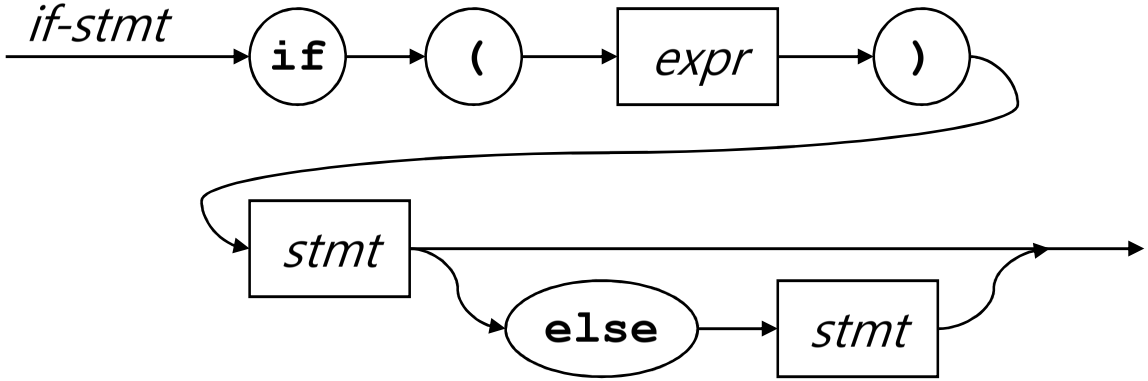
- EBNF의 대안
- 범례
  - 터미널은 타원
  - 논터미널은 직사각형
  - 시퀀싱을 위해서는 화살표
- 몇 가지
  - 초보자에게 쉬움
  - 잘 안쓰임; EBNF가 크기가 더 작고 간결함

신택스 다이어그램의 예

- 반복
  - EBNF: ***expr*** → ***term*** { **+** ***term*** }



- 옵션
  - EBNF: ***if-stmt*** → **`if ( `** ***expr*** **`)`** ***stmt*** **[** **`else`** ***stmt*** **]**



파싱 ← 중요

- 무엇이 파싱인가?
  - 프로그램의 신택스를 분석
  - 프로그램의 신택스 트리 구성
- 파싱의 종류
  - 탑다운(하향식) 파싱: 리커시브 디슨트 파싱 (recursive-descent parsing), LL 파싱
  - 바텀업(상향식) 파싱: 시프트-리듀스 파싱 (shift-reduce parsing)
- 파서 생성기
  - 파서 표현에서 파서를 생성하는 도구
    - YACC, Bison, BYACC, ANTLR, …

리커시브 디슨트 파서 (Recursive-Descent Parser)

- 중요
- 문법에서 파서를 손으로 만들었던 옛날의 방법
- 다음의 문법 변환이 필요함
  - 좌 재귀 제거 (left recursion removal)
  - 좌 팩토링 (left factoring)
- 구축 방법
  - 논터미널에 부합하는 재귀 프로시쥬어의 집합들을 만듦
  - 각 프로시쥬어의 동작은 상응하는 논터미널의 프로덕션의 RHS에 기반함
  - 각 토큰을 하나하나 읽어들여 정합하는 “match”라는 프로시쥬어가 있음

리커시브 디슨트 예

- 문법
  expr → ***term*** { + ***term*** }
  term → ***factor*** { * ***factor*** }
  factor → (***expr***) | ***number***
- 코드 스케치

```
expr()
{ term();
  while (token == '+')
  { match(token);
    term();
  }
}
```

```
factor()
{ if (token == '(')
  { match(token);
    expr();
    match(')');
  }
  else
    number();
}
```

파싱의 문제들

- 프리딕티브 파서
  - 하나의 룩어헤드(미리 보기; lookahead – 다음에 읽을 토큰) 심볼에 의해 어떤 동작을 할지를 정하는 파서
- First 와 Follow
  - First(α): 스트링 α의 처음 시작하는 토큰들의 집합
  - Follow(α) : 스트링 α 다음에 오는 토큰들의 집합
- 예
  - First((***expr***)) = { (}
  - For the grammar rule ***factor*** → (***expr***),) ∈ Follow(***expr***)

LL(1) 조건 ← 중요

- 프리딕티브 파싱의 조건
  - 두 개의 프로덕션 $A \rightarrow \alpha_1 \mid \alpha_2$ 에 대해 $First(\alpha_1) \cap First(\alpha_2)$ 는 반드시 공집합이어야 함
  - 만일 A 가 옵셔널이면, $First(A) \cap Follow(A)$ 는 공집합이어야 함
- 반례

  S → **`if`** ***C*** **`then`** ***S*** | **`if`** ***C*** **`then`** ***S*** **`else`** ***S***

  L → **`a`** ***L*** **`a`** | ε
- 더 자세한 부분은 컴파일러 구성에 대한 과목에서 다룸

스캐너/파서 생성기 ← 중요

- 컴파일러 구축 도구
  - 스캐너 생성기는 주어진 정규식 파일에서 스캐너 코드를 생성함
  - 파서 생성기는 주어진 문법 파일에서 파서 코드를 생성함
- 전형적인 예
  - 유닉스 툴 Lex (Lesk, 1975)와 YACC (Johnson, 1975) 이 제일 많이 쓰임
  - 최근의 공짜 버전은 Gnu Bison 과 Flex ("Fast lex").
  - 자바: JavaCC, JFlex/CUP, ANTLR, 등등
- YACC/Bison 스펙의 예 (Figure 4.13)
- 더 자세한 부분은 컴파일러 구성에 대한 과목에서 다룸

렉식스(Lexics),신택스, 시맨틱스

- 렉식스와 신택스 간의 명확한 경계가 없음
  - 토큰의 구조가 문법으로 설정될 수 있음
  - 구현에서 이러한 것에 대한 결정을 함
- 정적인 시맨틱스를 어떻게 봐야 할지에 대한 합의가 없음
  - 정적 시맨틱스는 CFG로 기술되지 않는 신택스로 볼 수 있음.
  - 정적 시맨틱스는 번역 시간에 결정되는 시맨틱스로 볼 수 있음. (이 교재의 관점).