BNF

차리서 <reeseo@konkuk.ac.kr>

건국대학교 공과대학 컴퓨터공학부

학기 전체 일정

주	월	화	수	목	금	토	일	실습	강의
1	8月 28	29	30	31	1	2	3	수강 정정, 팀 결성	과목 오리엔테이션, 팀 결성 안내
2	9月 4	5	6	7	8	9	10	팀 결성 완료, 주제 선정 시작	주제 선정 안내, 기획서 안내
3	11	12	13	14	15	16	17	주제 선정 완료, 기획 시작	기획서 안내
4	18	19	20	21	22	23	24	**기획서 작성**	**설계 문서 안내**
5	25	26	27	28	29	30	1	설계 문서 작성	(설계 문서 문답)
6	10月 2	3	4	5	6	7	8	설계 문서 작성	요구사항 분석 안내
7	9	10	11	12	13	14	15	요구사항 분석 및 재설계	(요구사항 분석, 재설계 문답)
8	16	17	18	19	20	21	22	요구사항 분석 및 재설계	구현 및 검사 안내, 중간 발표 안내
9	23	24	25	26	27	28	29	구현 및 검사	(구현 및 검사 문답)
10	30	31	1	2	3	4	5	구현 및 검사	(구현 및 검사 문답)
11	11月 6	7	8	9	10	11	12	중간 발표	—
12	13	14	15	16	17	18	19	요구사항 분석, 재설계/구현	(재설계/구현 문답)
13	20	21	22	23	24	25	26	요구사항 분석, 재설계/구현	(재설계/구현 문답)
14	27	28	29	30	1	2	3	요구사항 분석, 재설계/구현, 검사	(재설계/구현 문답)
15	12月 4	5	6	7	8	9	10	요구사항 분석, 재설계/구현, 검사	기말 발표 안내
16	11	12	13	14	15	16	17	기말 발표	—

이 수업에서의 BNF

기획서/요구분석서와 설계서에서:

- *사용하지 **않아도** 됨*
- **사용할 경우, 자연어와 병기해도 되고 자연어를 대체해도 됨**
  - 자연어와 병기할 경우 (권장): 자연어는 반드시 '맞아야'하고, BNF는 틀려도 됨
  - 자연어를 대체할 경우: 틀리면 안 됨!

구현에서 (파서[parser] 및 파서 생성기[parser generator]를):

- 사용하지 않아도 됨
- 사용할 경우, 맞으면 상당히 편하지만 틀리면 구현 불가
- 사용법이 매우 복잡하고 방대함

촘스키 위계Chomsky hierarchy

유형	문법grammar	언어language, set	인식하는 자동 기계
Type-0	Unrestricted Grammar 무제약 문법	Recursively Enumerable set 재귀 열거 언어	Turing machine 튜링 머신
Type-1	Context-Sensitive Grammar 문맥 의존 문법	Context-Sensitive Language 문맥 의존 언어	Linear-bounded non-deterministic Turing machine 선형경계 비결정적 튜링 머신
Type-2	**Context-Free Grammar** **문맥 자유 문법**	**Context-Free Language** **문맥 자유 언어**	**Non-deterministic pushdown automaton** **비결정적 푸시다운 오토마타**
Type-3	Regular Grammar 정규 문법	Regular Language 정규 언어	Finite state automaton 유한 상태 기계

- 문맥 자유 언어의 예: $\{ \mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N} \} = \{ \varepsilon, \mathtt{ab}, \mathtt{aabb}, \mathtt{aaabbb}, \dots \}$
  → a들을 확인하면서 그 a들이 **몇 번 나왔는지 기억하고 있어야** 그 횟수 만큼 b가 나오는지 확인 가능
- 푸시다운 오토마타에는 **기억장치**(스택)가 있다!
- 대다수 프로그래밍 언어의 문법은 문맥 자유 문법

Backus-Naur Form (BNF)

문맥 자유 문법을 표현하기 위한 표기법

- “Backus-Normal Form”이라고도 함
- 표현식에 이름을 붙이고, 재귀적으로 부를 수 있음: 기억장치 역할

문맥 자유 문법의 파서를 생성해주는 도구[parser generator] 를 위한 입력 언어

- 도구 예시: Yacc, GNU bison, JavaCC, Happy 등
- 각 도구마다 세부 문법 상이 (큰 틀만 비슷)

다양한 변종이 존재

- Extended BNF (EBNF), Augmented BNF (ABNF), Parsing Expression Grammar (PEG) 등
- 오히려 BNF는 확고한 표준이 없음; 반면에 EBNF 등 일부 변종들은 표준 존재
- 파서 자동 생성 목적보다 사람의 가독성과 분석 목적에 주안점을 둔 변종도 존재
- 이 과목에서는 기본 BNF에 약간의 편의 기능을 추가하여 소개

BNF: 다짜고짜 예제부터 (for your 감)

정규 표현식

```
-?(0|[1-9][0-9]*)(\.[0-9]+)?
```

- 음수 부호는 없거나 하나만 있고, 0에도 음수 부호가 붙을 수 있음 (“-0”, “-0.00” 등 허용)
- 정수부는 최소 한 자리 이상이며, 정수부가 0인 경우만 “0”이고 그 외의 선행 0은 금지함
- 소수부는 “.”으로 시작하고 뒤에 최소 하나 이상의 숫자가 따라 나오며, 후행 0은 허용함

이를 BNF로 (“number”라는 이름을 붙여서) 나타내면:

```
          <number> ::= <sign> <integer> <decimal>
            <sign> ::= "" | "-"
         <integer> ::= "0" | <non-zero-digit> <digits>
         <decimal> ::= "" | "." <digit> <digits>
          <digits> ::= "" | <digit> <digits>
           <digit> ::= "0" | <non-zero-digit>
<non-zero-digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

(자연어로 느슨하게 설명하는) 기본 BNF 문법 [1/2]

(이 페이지에 등장하는 각종 “구분자”의 좌우에는 0 개 이상의 공백이나 탭이 추가될 수 있음)

- BNF 전체 정의는 최소 하나 이상의 개행으로 서로 구분된, 한 개 이상의 “룰”들의 나열
- 각 룰은 문자열 “`::=`”를 주 구분자로 삼아, 그 좌변(룰 이름)과 우변(표현식)으로 구성

  `<decimal> ::= "" ¦ "." <digit> <digits>`
- 룰 이름은 비단말non-terminal 기호. (다음 페이지에서 설명)
- 같은 룰 이름을 갖는 둘 이상의 룰들이 존재할 수 **없음**
- 표현식은 한 개 이상의 선택지choice 들의 (“¦”로 서로 구분된) 나열

  `"" ¦ "." <digit> <digits>`
- 선택지는 한 개 이상의 기호symbol 들의 (공백으로 서로 구분된) 나열

  `"." <digit> <digits>`

(자연어로 느슨하게 설명하는) 기본 BNF 문법 [2/2]

- 기호에는 “`::=`”의 좌변에도 등장하는 비단말 기호와 좌변에는 등장하지 않는 단말terminal 기호가 있음
- 비단말 기호는 문자 “<”와 “>”로 둘러싸인 “이름”

  <decimal>, <digit> 등

  - 이름은 로마자 대소문자로 시작하고, 그 뒤에 0개 이상의 로마자 대소문자나 숫자, ‘-’ 문자들로 구성
- 단말 기호는 문자열 리터럴:[1)]
  - 큰 따옴표가 포함되지 않은 문자열은 큰 따옴표로 둘러싸고, 큰 따옴표가 포함된 문자열은 작은 따옴표로 둘러쌈.
  - 둘 다 포함된 문자열은 분리해서 나란히 놓으면 됨 (`'He said, "' "She's" 'gone!"'`)

[1)] 개행 문자 <EOL>과 탭 문자 <TAB>은 (명시적으로 좌변에 등장하지는 않더라도) 전역에 정의된 비단말 기호로 보는 경우가 대부분이지만, 간혹 단말 기호에 포함시키기도 함.

정규 표현식과 BNF 비교

정규식	의미	BNF
`(abc)?`	0 번 혹은 1 번	`<opt-abc> ::= "" | "abc"`
`(abc)*`	0 번 이상	`<opt-abcs> ::= "" | "abc" <opt-abcs>`
`(abc)+`	1 번 이상	`<abcs> ::= "abc" | "abc" <abcs>`
`(abc){1,3}`	1 번 이상 3 번 이하	`<abcs1to3> ::= "abc" | "abcabc" | "abcabcabc"`
`[a-e]`	정해진 문자들 중 하나	`<sel-ch> ::= "a" | "b" | "c" | "d" | "e"`
`[a-e]*`	선택 반복	`<sel-rep> ::= "" | <sel-ch> <sel-rep>`
`.`	아무 문자 하나	`<sel-any> ::= "a" | "b" | ...` (개행 빼고 모두 나열)
`[가-힣]`	아무 한글 하나	`<hangeul> ::= "가"|"각"|"갂" ... "힡"|"힢"|"힣"`

"에이, 정규식이 훨씬 간편하네요. BNF 필요 없네요."

정규식으로는 못 하지만 BNF로는 할 수 있는 것들

Q: 문자 a가 0 번 이상 n 번 나오고, b가 똑같이 n번 나오는 문자열? (언어 $\{a^n b^n \mid n \in \mathbb{N}\}$의 원소)

A: 정규식으로는 안 되지만, BNF로는 아래처럼 하면 됩니다:

```
<S> ::= "" | "a" <S> "b"
```

Q: 그러니까, 그런 특수한 경우를 어디에 쓰냐구요?

A: 예를 들어, 사칙연산 수식을 처리할 때 씁니다. 참고로, 아래에서 <number>는 앞서 '다짜고짜 예제' 페이지에 나왔던 그 <number>입니다:

```
<expression> ::= <term> | <expression> " " <plusminus> " " <term>
      <term> ::= <factor> | <term> " " <timesdiv> " " <factor>
    <factor> ::= <number> | "(" <expression> ")"
<plusminus> ::= "+" | "-"
 <timesdiv> ::= "*" | "/"
```

예를 들어, 다음 식은 위 <expression> 규칙에 부합하는 올바른 수식 문자열입니다:

```
5.3 * (6.5 - (10 + -3.1) / 2) + 72 * 8.0
```

전기프 BNF

...라고 아무리 예시까지 들며 주장해봤자, 결국:

불편한 것은 불편한 것이다!

솔직히 강사도 이 '기본 BNF'는 불편해서 별로 쓰고 싶지 않음.

- 이 과목의 문서에서, 정규식만으로 간단히 표현 가능한 요소는 그냥 정규식으로 표현하기를 권장[2)]
- 이름을 붙이(고|거나) 재귀적으로 표현해야할 때에는 지금부터 제안하는 '전기프 BNF' 사용 권장
- 전기프 BNF의 확장 방식은 실존하는 많은 변종 BNF들이 흔히 공통적으로 사용하는 방식 + α
  (가능한 한 적고 간결한 확장만으로 많은 효과를 얻도록 조정)

전기프 BNF의 확장 내용 핵심 요약:

- 따옴표 속은 (리터럴이 아니라) 항상 정규식
- 기존의 기호$^{\text{symbol}}$ = {단말, 비단말}을 → 기호 = {단말, 비단말, 괄호 그룹}으로 변경
- (방금 추가한 괄호 그룹을 포함하여) 모든 기호에 반복 메타 문자 (?, * 등) 사용 가능
- 비단말 기호의 이름에 한글도 (첫 글자로도) 사용 가능
- 기타: 표현식 속 개행 및 주석 허용

2) 교육 (훈련) 상의 이유도 있음: BNF는 컴파일러/인터프리터 구현 시 매우 중요하지만 그 상황 외에는 사용할 일이 드문 반면, 정규표현식은 평생 옆에 끼고 살면서 약방의 감초처럼 수시로 (심지어 텍스트/코드 편집기에서도) 사용하게 될 가능성이 높음.

전기프 BNF 확장 사항 #1: 따옴표 속은 정규식

따옴표 속은 이제 리터럴 문자열이 아니라 **항상 정규식**!

기존 (지난 수업 때 배운) 정규식의 모든 요소를 그대로 똑같이 사용. 단:

- 큰 따옴표 속에서, 큰 따옴표 리터럴 문자는 `\"`로, 작은 따옴표는 리터럴 문자는 `'`로 표기
- 작은 따옴표 속에서, 큰 따옴표 리터럴 문자는 `"`로, 작은 따옴표는 리터럴 문자는 `\'`로 표기

주의 사항:

- <EOL>과 <TAB> 존재 의미 상실: 각각 `"\n"`과 `"\t"`로 (혹은 `'\n'`과 `'\t'`로) 대체 사용 권장
- 역으로 다소 불편해진 점: 이제 소수점은 `"."` 대신 `"\."`으로, 덧셈 기호는 `"+"` 대신 `"\+"`로 표기해야 함

예시:

기존: `<non-zero-digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"`

확장: `<non-zero-digit> ::= "[1-9]"`

전기프 BNF 확장 사항 #2: 괄호 그룹

표현식 속(“::=” 우변)의 **단말/비단말 기호**가 등장하던 모든 자리에, **괄호로 둘러싸인 그룹**도 등장 가능

문법적으로:

- 괄호 **외부**에서, 괄호 전체를 **하나의 BNF 기호**로 (즉, 단말/비단말 기호와 동격으로) 취급
  - → 다음 페이지에 설명할 ‘반복’의 대상에 포함됨
- 괄호 **내부**에서, 괄호 전체를 **(재귀적으로) 표현식**으로 (즉, “::=”의 우변 전체처럼) 취급
  - → 문법 조건도 표현식과 동일: 괄호 속에는 하나 이상의 선택지가 있어야 하고 각 선택지마다 하나 이상의 기호가 있어야 함
  - → 따라서 괄호는 중첩될 수 있음 (괄호 내부가 표현식이므로, 기호인 괄호가 그 안에 등장 가능)

예시 (동치인 표현식들):

```
A B | (C D | E F) | G H  =  A B | C D | E F | G H
A B | C (D | E) F | G H  =  A B | C D F | C E F | G H
A B | C ((D | E) F | G) H  =  A B | C D F H | C E F H | C G H
```

전기프 BNF 확장 사항 #3, #4: 반복 메타 문자와 한글 이름

표현식 속(“::=” 우변)의 **모든 기호**에 정규식 **반복 메타 문자** 사용 가능:

`?` `*` `+` `{n}` `{n,m}` `{n,}`

예시:

기존: `<digits> ::= "" | <digit> <digits>`

확장: `<digits> ::= <digit>*` (물론 “`<digits> ::= "[0-9]*""`”이 더 직접적)

비단말 기호의 이름에 (첫 글자로도) **한글** 사용 가능:

`[a-zA-Z가-힣][a-zA-Z가-힣0-9-]*`

예시:

- `<숫자들> ::= "[0-9]*"`
- `<표현식> ::= <항> | <표현식> "␣" <더하기빼기> "␣" <항>`
  - → 물론, 이것도 가능: `<표현식> ::= (<항> "␣[\+-]␣")* <항>`
  - → 연산자 좌우 공백 자유: `<표현식> ::= (<항> "␣*[\+-]␣*")* <항>`

전기프 BNF 확장 사항 #5: 표현식 속 개행

먼저, 이 페이지에서 말하는 “공백”은 스페이스 바 문자와 탭 문자임.

표현식 속(“::=” 우변)의:

- 왼쪽에 기호가 있고 (사이에 공백들을 두고) 오른쪽에 선택지 구분자 “¦”가 있을 때, 그 사이
- 두 기호 사이

에 개행 (및 그 개행문자 좌우에 임의의 갯수의 공백들) 추가 가능

단순히 문서 가로 폭에 의한 자동 줄바꿈보다는 그나마 가독성을 높이려는 의도. 예시:

```
<지번> ::= <한글>+ "시␣+" <한글>+ "구␣+" <한글>+ <양수>? "동␣+" <양수> ("-" <양수>)?
        ¦  <한글>+ "도␣+" <한글>+ "시␣+" <한글>+ "면␣+"
           <한글>+ <양수>? "리␣+" "산␣+"? <양수> ("-" <양수>)?
        ¦  <한글>+ "도␣+" <한글>+ "군␣+" <한글>+ "면␣+"
           <한글>+ <양수>? "리␣+" "산␣+"? <양수> ("-" <양수>)?
<양수> ::= "[1-9][0-9]*"
<한글> ::= "[가-힣]"
```

전기프 BNF 확장 사항 #6: 주석

따옴표 속이 아닌 곳에 등장한 “;” 문자부터 그 줄의 끝까지는 주석

- 따옴표 속의 “;”는 그냥 (정규식의) 리터럴 문자
- 주석 속에는 개행문자 이외의 모든 문자 등장 가능

```
; 음수 부호는 옵션, 0에도 음수 부호 가능 ("-0", "-0.00" 등 허용)
; 정수부 한 자리 이상, 선행 0 금지
; 소수부 "."으로 시작, 최소 하나 이상의 숫자, 후행 0 허용
       <number> ::= <sign> <integer> <decimal>
         <sign> ::= "" | "-"                  ; 사실은 <opt-sign>이 더 적절
      <integer> ::= "0" | <non-zero-digit> <digits>
      <decimal> ::= "" | "." <digit> <digits>
       <digits> ::= "" | <digit> <digits> ; 숫자'들'
        <digit> ::= "0" | <non-zero-digit>
<non-zero-digit> ::= "1" | "2" | "3" | "4" ; 양수만 필요한 경우 있음
                   |  "5" | "6" | "7" | "8" | "9"
```

전기프 BNF: 예제 (사칙연산 산술식) [1/2]


```
<expression> ::= <term> ¦ <expression> " " <plusminus> " " <term>
      <term> ::= <factor> ¦ <term> " " <timesdiv> " " <factor>
    <factor> ::= <number> ¦ "(" <expression> ")"
<plusminus> ::= "+" ¦ "-"
 <timesdiv> ::= "*" ¦ "/"
    <number> ::= <sign> <integer> <decimal>
      <sign> ::= "" ¦ "-"
   <integer> ::= "0" ¦ <non-zero-digit> <digits>
   <decimal> ::= "" ¦ "." <digit> <digits>
    <digits> ::= "" ¦ <digit> <digits>
     <digit> ::= "0" ¦ <non-zero-digit>
<non-zero-digit> ::= "1" ¦ "2" ¦ "3" ¦ "4" ¦ "5" ¦ "6" ¦ "7" ¦ "8" ¦ "9"
```

⇩

```
<표현식> ::= (<항> " [\+-] ")* <항>
    <항> ::= (<인수> " [\*/] ")* <인수>
  <인수> ::= <수> ¦ "(" <표현식> ")"
    <수> ::= "-?(0¦[1-9][0-9]*)(\.[0-9]+)?"
```

전기프 BNF: 예제 (사칙연산 산술식) [2/2]

- 연산자와 수, 괄호 사이에 공백을 자유롭게 (0 개 이상 임의의 갯수로) 허용

```
<표현식> ::= (<항> "␣*[\+-]␣*")* <항>
  <항> ::= (<인수> "␣*[\*/]␣*")* <인수>
<인수> ::= <수> ¦ "␣*(␣*" <표현식> "␣*)␣*"
  <수> ::= "-?(0¦[1-9][0-9]*)(\.[0-9]+)?"
```

- 전체 정의문 중 우변에서는 (<인수> 룰의 우변에서) 한 번 밖에 안 쓰이는 <수> 룰은 <인수> 룰 내부에 넣어버릴 수 있음

```
<표현식> ::= (<항> "␣*[\+-]␣*")* <항>
  <항> ::= (<인수> "␣*[\*/]␣*")* <인수>
<인수> ::= "-?(0¦[1-9][0-9]*)(\.[0-9]+)?" ¦ "␣*(␣*" <표현식> "␣*)␣*"
```

참고: BNF와 Parser, 그리고 Parse Tree [1/3]

산술식 초기 예제 (이하 “A”)

```
<expression> ::= <term> | <expression> " " <plusminus> " " <term>
      <term> ::= <factor> | <term> " " <timesdiv> " " <factor>
    <factor> ::= <number> | "(" <expression> ")"
<plusminus> ::= "+" | "-"
 <timesdiv> ::= "*" | "/"
```

를 왜 다음(이하 “B”)과 같이 하지 않았을까?

```
<expression> ::= <operand> | <expression> " " <operator> " " <operand>
   <operand> ::= <number> | "(" <expression> ")"
  <operator> ::= "+" | "-" | "*" | "/"
```

참고: BNF와 Parser, 그리고 Parse Tree [2/3]

파서 생성기에 문법 넣으면 파서 생성 → 파서에 문자열 넣으면 (문법에 맞을 경우) 파스 트리 생성

- 파서 생성기 G에 BNF 정의문 A, B를 각각 넣으면 파서 P_A, P_B가 각각 만들어짐: G(A) = P_A, G(B) = P_B
- P_A, P_B에 같은 수식 E를 넣으면 (문법에 맞을 경우) E에 대한 각 문법 기준 파스 트리 생성
- 모든 E에 대해, P_A, P_B의 파스 트리 생성 여부는 동일 (즉, 문법 부합 검사 결과는 A, B 동일)
- 단, 파스 트리 내용은 서로 다를 수 있음: $P_A(E) \neq P_B(E)$인 E가 존재
- A는 *, /의 결합 강도가 +, -보다 높음. B는 네 연산자 결합 강도 동일.

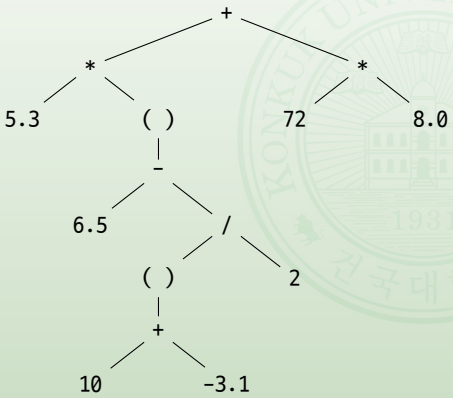
이 과목에서는

- 정규 표현식: 문서 뿐만 아니라 구현에도 (가능할 경우) 정규식 라이브러리 사용 권장
- BNF: 문서에만 (필요할 경우에만) 사용. (즉, 구현시 파서 생성기 사용은 권장하지 않음)

참고: BNF와 Parser, 그리고 Parse Tree [3/3]


```
E = 5.3 * (6.5 - (10 + -3.1) / 2) + 72 * 8.0
```

$P_A(E)$



$P_B(E)$