3주차

2022년 3월 8일 화요일 오후 3:33

3주차 syntax

Ppt보면서 이해할수있는것은 넘어가고 중요한 개념만 짚고 넘어 가겠다

Kleen star : L에서 만들수있는 모든길이의 string들을 합친것이다 empty포함

Dager: kleen star - empty

Grammer

G = (Vn, Vt, P, S)

Vn : nonterminal symbols의 유한집합
 문자열 생성시 사용되는 중간과정의 기호, 보통 대문자
 nonterminal은 언어에 대한 계층 구조 부여 가능
Vt: terminal symbols의 유한집합
 언어의 정의된 기호로, 영어 소문자, 아라비아 숫자, 연산자, 기호등을 포함
Vn ∩ Vt = 공집합

V_N : non teminal symbols 대문자 ABC..
V_T : terminal symbols 소문자 abc..

-> 서로 공집합 없다

P : productions -> rule set
S : start symbol non terminal symbol이다

Greek 문자등은 kleen star of V에 속하는 string

G = {{V_N},{V_T},P,S}

Derivation

직접 유도 : 생성 규칙을 적용한다는 의미

간접 유도: 직접 유도를 0번 또는 그이상의 step으로 하는것을 의미한다

+간접유도는 적어도 한번 이상 직접 유도를 한다

Leftmost derivation :가장 왼쪽 symbol 치환하면서 진행
Rightmost derivation : 가장 오른쪽 symbol 치환하면서 진행

Generation

만약 a1이 a2를 간접 유도 했다면 a1은 a2를 생성했다고 한다

Grammar는 Language를 생성하고
Language는 Grammar를 degin한다

Equivalent Grammars 동치

생성하는 결과물 같으면 동치라고 본다

Type of Grammars

1. Type 0: No restrictions
2. Type 1 : Context sensitive grammar (CSG)
    a. 특정 문맥에서만 변환가능
3. Type 2 : Context free grammar
    a. 문맥 관계 없이 언제나 변환 가능
4. Type 3 : Regualar grammar
    a. String이 한쪽 방향으로만 자랄수있다

Type 1,2,3문법으로 만든 language가 따로 있다

4주차 syntax 2

2022년 3월 23일 수요일 오후 12:18

Arithmetic Expression Grammar

예) $2-(3+4)$ 의 유도

$E \rightarrow E+E \mid E \rightarrow E-E$
$E \rightarrow (E)$
$E \rightarrow 0 \mid \ldots \mid 9$

$E \Rightarrow E-E$
$\Rightarrow 2-E$
$\Rightarrow 2-(E)$
$\Rightarrow 2-(E+E)$
$\Rightarrow 2-(3+E)$
$\Rightarrow 2-(3+4)$

연습
$(5-3)-(2-(3+4))+6$

화면 캡처: 2022-03-23 오후 10:55

Parse Trees

문장 (sentence)의 유도단계를 표현한 트리
유도 관계를 트리로 표현

- **root node**는 **start symbol**
- **Internal node**는 **nonterminal symbol**
- **Leaf node**는 **terminal symbol**
- 생성된 string은 모든 leaf nodes를 연결하여 얻는다 (left -> right)

Integer Grammar

예) 352

Integer → *Integer Digit* | *Digit*
Digit → 0 | ... | 9

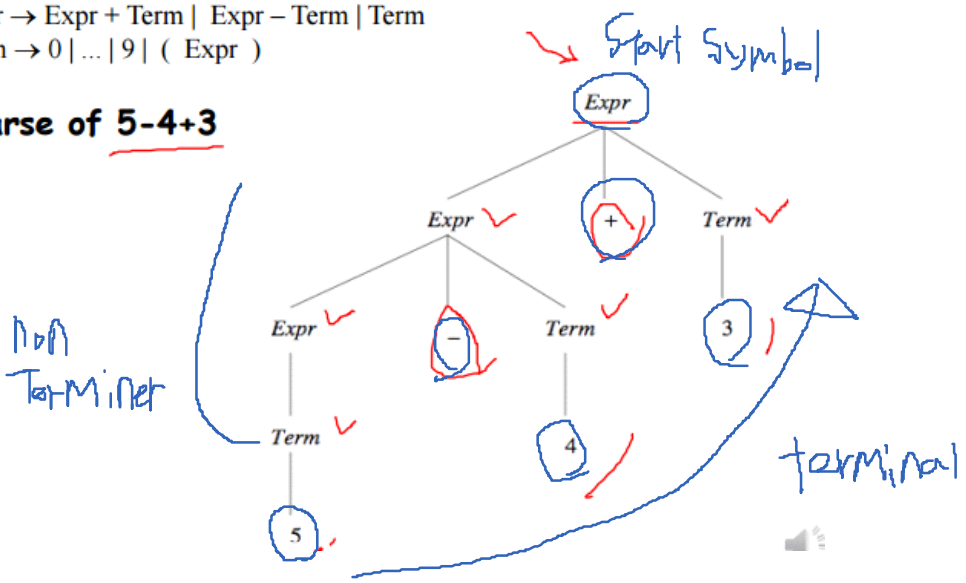
Integer => *Integer Digit*
=> *Integer Digit Digit*
=> *Digit Digit Digit*
=> 3 *Digit Digit*
=> 3 5 *Digit*
=> 3 5 2

화면 캡처: 2022-03-23 오후 10:57

Arithmetic Expression Grammar

Expr → Expr + Term | Expr – Term | Term
Term → 0 | ... | 9 | (Expr)

Parse of 5-4+3



화면 캡처: 2022-03-23 오후 11:01

Associativity and Precedence (결합성과 우선순위)

계산 진행 방향 : 좌결합

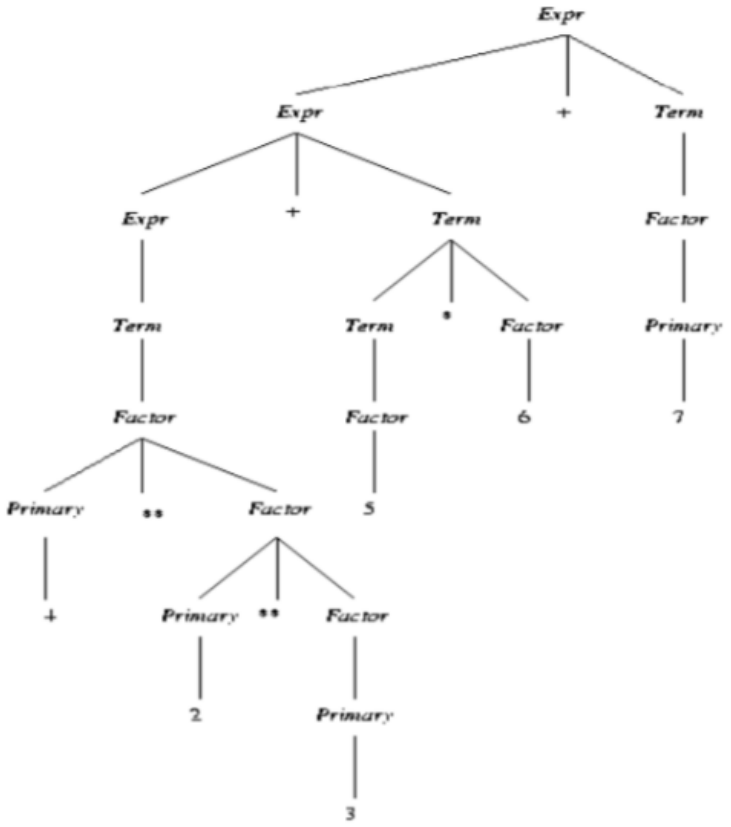
*/가 +-보다 우선순위 높다

Grammar G_4 :

Expr -> Expr + Term | Expr - Term | Term
Term -> Term * Factor | Term/Factor | Term % Factor | Factor
Factor -> Primary ** Factor | Primary
Primary -> 0|....|9 | (Expr)

Parse Tree of Grammar G_4

화면 캡처: 2022-03-23 오후 11:08



화면 캡처: 2022-03-23 오후 11:08

Operation 우선 순위 있다

**은 우결합하고 좌결합 진행한다

Parse tree는 left most deribation인지 right most derivation인지 알 수 없다

Parse tree에서 아래 쪽에 있을수록 우선 순위가 높아진다

같은 심볼이 있는 방향에 따라 recursion과 associativity 결정된다

Left associativity results from left-recursion

A -> AB - left recursion
F -> P**F - right recursion

Ambiguous Grammar

같은 senctence에 대해 두가지 이상의 parse tree가지는 문법

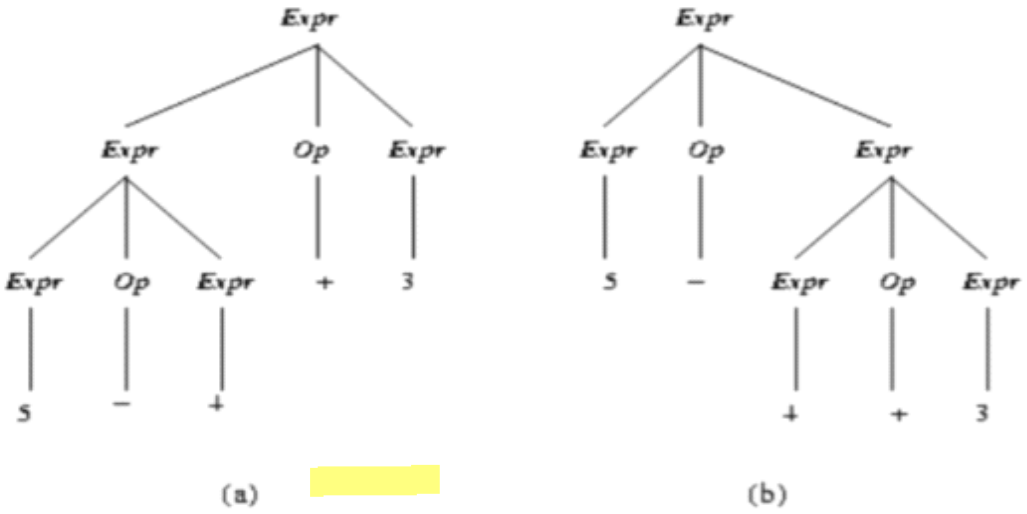
동치 표현

- 문장 A가 두개의 다른 parse tree가진다
- <=> 문장 A가 두개의 다른 leftmost derivations를 가진다
- <=> 문장 A가 두개의 다른 rightmost derivations를 가진다

Leftmost derivations와 rightmost derivations 같이 쓰면 안된다

예: G_3 : Expr -> Expr Op Expr | (Expr) | 0 | 1 | 2|..| 9
Op -> + | - | * | / | % | **

5 - 4 + 3의 파스 트리는?



5 – 4 + 3의 두 파스 트리

a = 4 / b= -2

화면 캡처: 2022-03-23 오후 11:18

If and the Dangling Else

조건

IfStatement -> if (Expression) Statement | if (Expression) Statement else Statement

Statement -> Assignment | IfStatement | Block

Block -> {Statement_2}

Statement_2 -> Statement_2 Statement | Statement

EX)

문장 : If(e1) if (e2) S1 else S2

1. If (e1)을 rule 1으로 보고 나머지를 rule2로 적용하는 방법
   -> If(e1) if (e2) S1 else S2

2. If (e1) else S2를 rule2로 적용하고 나머지를 rule1으로 적용하는 방법
   -> If(e1) if (e2) S1 else S2

어떤걸 채택해야하나?
자바에서는 가장 가까운 if문에 걸리게 함

Ambituity의 해결 방법

1. 문법을 처음부터 모호성이 없도록 설계 한다
   a. 문법이 길고 복잡해질수있다
   b. EX) G3 : Expr -> Expr OP Expr | (Expr)
   c. OP -> +| - | * | % | **

      G4 : Expr -> Expr + Term | Expr-Term|Term
      Term -> Term * Factor | Term/Factor|Term % Factor | Factor
      Factor -> Primary**Factor | Primary
      Primary -> I | (Expr)

   d. G3 <=> G4이지만 G3은 모호한 문법이다

2. Grammar 외에 추가 규칙을 사용한다
   a. 항상 직전에 있는 if문에 속한다
   b. 필요할 경우 {}로 if문의 경계 명확히 함

Extended BNF (EBNF)

BNF : 반복문을 위한 재귀 있음
EBNF : 추가 기호를 넣어 재귀를 없앰

{} : 0번 이상 반복 될수있어야함 - iteration
() : 문조건 하나를 골라야함 - Choice
[] : 하나를 골라도 되고 안골라도 됨 - Option

EX)

BNF : Expr -> Expr + Term | Expr-Term | Term

-> Expr은 +/-로 구분되는 하나이상의 Term 이다

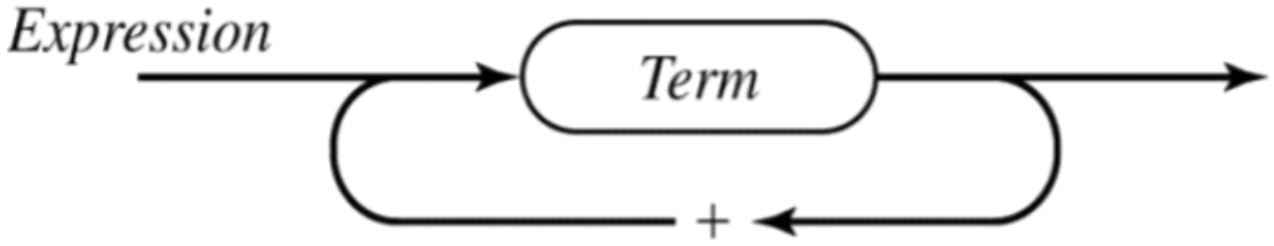
EBNF : Expr-> Term {(+|-) Term}

If statment

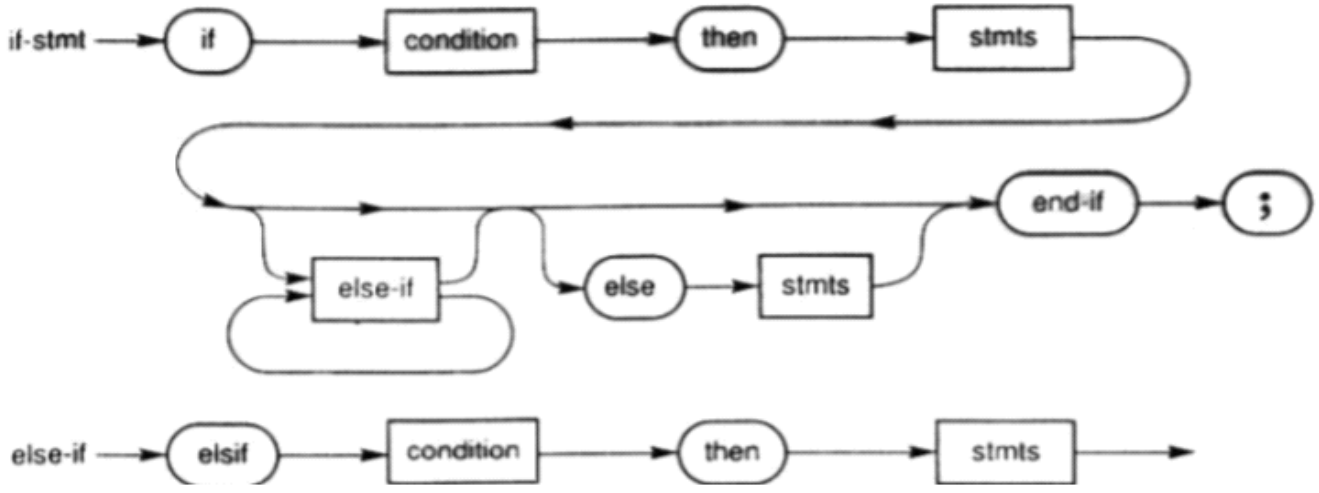
BNF: IfStatement –> if (Expression) Statement | if (Expression) Statement else Statement

EBNF: if (Expression) Statement [else Statement]

EBNF to BNF : 서로 상호 변환 가능하고 표현력이 같다



Syntax Diagram for Expressions with Addition



Syntax Diagram for if-stmt

화면 캡처: 2022-03-23 오후 11:44

동그라미 : Terminal symbol
사각형 : Non Terminal symbol

An Example program written in myFunction

과제 실습 4를 같이 참고하면서 봐라

BNF와 EBNF표현을 같이 볼것이다

unary −, !
exponentiation **
multiplication *, /, %
addtion +, −
Relation >, <, >=, <=
equality ==, !=
Conjunction &&

연산자 우선순위로 올라갈수록 높아진다

My Function in BNF

Program -> int main() {Declatrations Statement_2}

Declarations -> Declarations Decl | ε

Decl -> Type IdList

IdList -> IdList Id | Id

Statement_2 -> Statement_2 Statement | Statement

Statement -> ; | Assignment | IfStmt | Block

Assignment -> Id = Expr

IfStmt -> if (Expr) Statement | if (Expr) Statement else Statement

Block -> {Statement_2}

Type -> int | float

Expr -> Expr && RelExpr | RelExpr

RelExpr -> AddExpr RelOp AddExpr | AddExpr

AddExpr -> AddExpr + Term | AddExpr-Term | Term

Term -> Term *Factor | Term /Factor | Factor

Factor -> Primary **Factor | Primary | unaryOp Primary

Primary -> Num | Id | (Expr)
RelOp -> < | > | <= | == | !=
unaryOp -> - | !

Id -> Letter LetterDigit
LetterDigit -> LetterDigit Letter | LetterDigit Digit | ε
Letter -> a | b|...|Z
Digit -> 0|...|9

Num -> Integer | Float
Integer -> Integer Digit | Digit
Float -> Integer . Integer

2.3 EBNF for myC

Program -> int main ( ) { Declarations Statements }
Declarations -> { Decl }
Decl -> Type Id { , Id }
Statements -> Statement { Statement }
Statement -> ; | Assignment | IfStmt | Block
Assignment -> Id = Expr;
IfStmt -> if (Expr) Statement [else Statement]
Block -> { Statements }
Type -> int | float
Expr -> RelExpr { && RelExpr }
RelExpr -> AddExpr RelOp AddExpr
AddExpr -> Term { (+ | -) Term }
Term -> Factor { (* | /) Factor }
Factor -> Primary { ** Primary } | UnaryOp Primary
Primary -> Num | Id | (Expr)
RelOp -> < | > | <= | == | !=
UnaryOp -> - | !

화면 캡처: 2022-03-24 오전 12:00

Lexical Level

Id -> Letter { Letter | Digit }
Letter -> a | b | ... | z | A | B | ... | Z
Digit -> 0 | 1| ... | 9
Num -> Integer | Float
Integer -> Digit { Digit }
Float -> Integer . Integer

화면 캡처: 2022-03-24 오전 12:00

5주차

2022년 3월 29일 화요일 오후 11:36

Clite

MyC BNF,EBNF는 4주차 가서 확인해봐라

EBNF 간단 리뷰

{} : 0번 이상 반복
[] : 하나 골라도 되고 안골라도 되고 option
() : 무조건 하나 선택 -> choice

- C언어의 subset 즉 일부분이다
- Page 문법에 맞는 간단한 문법이다

Clite Grammar : Statements

Program -> int main() {Declaration2 Statement2} // program 시작할때 main함수에 Declaration2와 Statement2를 선언한다

Declaration2 -> {Declaration} //Declatraion2는 Declaration을 0번 이상 사용 할 수 있다

Declaration -> Type Identifier [[Integer]]{,Identifier[[Integer]]}; //이 선언문은 배열도 선언 할 수 있다
// arr[20], c[3], b, A[20] = {4,5,6,..} , G[34] = {arr[3],4,...}

Type -> int | float | bool | char //type은 int float bool char 4가지 타입 있다

Statement2 -> {Statement} //Statement2는 Statement를 0번 이상 반복 할 수 있다

Statement -> ; | Block | Assignment | IfStatement | WhileStatement //Statement는 ; , Block , Assigment등 을 호출할수있다
// ; 단독으로 지원 되므로 null -statement를 지원한다는것을 알 수 있다

Block -> {Statement2} //Block에는 Statement를 0번 이상 호출할수있는 Statement2를 호출할수있다

Assignment -> Identifier [[Expression]] = Expression; // 변수 할당 뿐 아니라 배열도 할당할수있다

IfStatement -> if (Expression) Statement2 [else Statement2] // If 문을 정의한것이다

//Statement2같은데 나중에 알아보세요
WhileStatement -> while(Expression) Statement2 //while 문 정의

Conjunction -> Equality {&& Equality} //Equality의 && 식

Equality -> Relation[EquOp Relation] // == , !=로 판단 할 수 있는 식

EquOp -> == | != //같은지 다른지 연산자를 지원한다 연산자의 우선 순위를 구분한것이다

Relation -> Addition [RelOp Addition] //

RelOp -> < | > | <= | >= //연산자 우선순위에 따라 EquOp와 구분한것이다

Addition -> Term { AddOp Term}

AddOp -> + | -

Term -> Factor {MulOp Factor}

MulOp -> * | / |%

Factor -> [UnaryOp] Primary //

UnaryOp - > - | !

Primary -> Identifier [[Expression]] | Literal | (Expression) | Type(Expression) //arr[10], arr[brr[4]] , A , (Val) ,int(s) -> c랑은 다른 typecasting을 지원한다

Expression -> Conjunction{ || Conjunction} // OR 연산자를 지원할수있다 Conjunction을 or로 표현한 식

Identifier -> Letter { Letter | Digit} //문자와 숫자의 조합 일종의 문자열로 생각하면 될듯

Letter -> a | b | ……| A | B |….| Z

Digit -> 0 |1…|9

Literal -> Integer | Boolean | Float | Char //자료형의 값을 표현하는 식

//각 타입의 값 표현 식
Integer -> Digit {Digit}
Float -> Integer . Integer
Boolean -> true | false
Char -> 'ACII Char' //character 로 상수 'a'도 표현할수있다

Clite operator 우선 순위

Clite Operator	Associativity
Unary - !	none
* /	left
+ -	left
< <= > >=	none
== !=	none
&&	left
||	left

우선순위와 결합성

화면 캡처: 2022-03-30 오전 11:55

위의 clite statement와 grammar보면 우선순위에 따라서 parse tree들어가는것 같다

Clite에 없는 C 특징

- 주석
- 함수
- 다차원 배열
- For 문
- Case문
- Go to

- 초기화 intialize
- Enum
- Typedef
- 구조체
- Conditional expr (a>b) ? 1 : 0 삼항 연산자
- ++ , -- , -=, +=
- Pointer
- Sizeof
- Shift operation >> <<

6주차 Lexical

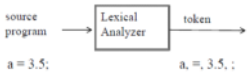
2022년 4월 6일 수요일 오후 11:42

Lexical Analysis (Scanner)

★ 전체적인 개념 및 연습개념 포함

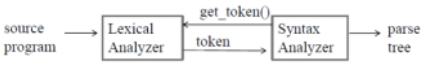
★ 지금까지의 내용 정리 한 느낌

어휘 분석기

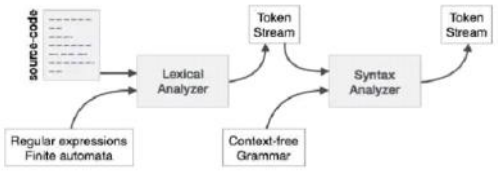


화면 캡처: 2022-04-06 오후 11:49

Source program에서 token을 추출하는 구문을 의미한다



화면 캡처: 2022-04-07 오전 11:43



Parser는 token을 이용한 parse tree build한다
Parser는 token 필요하면 lexer를 통해 처리를 한다 -> 가볍다,빠르다

Lexical module

lexical은 parser의 sub로 자리 잡고 있다

parser(syntax analysis)와 scanner(lexical analysis) 구분이유

1. Token는 Regular languages이다
   a. Regular languages는 context free language보다 간단한 방법으로 정의할수있다
   b. Lexier 즉 정규언어 인식기는 parser와 분리 하여 쉬운 방법으로 구현 할수있다. => 컴파일러 fornt-end의 모듈화 = Lexer + parser
2. Lexical 하는 일 parser가 다 할수있지만 모듈화 문제, token 구문구저 정의가 합쳐지는데서 오는 문제 , 프로그램이 커지는 문제

Token

문장에서 사용되는 최소 단위의 문법 요소 - **Terminal symbol**

예) if (x > y) x = 10 ;
(32,0) (7,0) (4,x) (25,0) (4,y) (8,0) (4,x) (23,0) (5,10) (20,0)

화면 캡처: 2022-04-07 오전 11:17

(Token name, Token value)

Token 정의

- Keyword
- Constants
- Identifiers
- String,integer,float ,double...
- Operator ...

int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

Category	Regular Expression
Keyword	bool | char | else | false | float | if | int | main | true | while
Identifier	letter (letter | digit)*
integerLit	digit+
floatLit	digit +.digit +
charLit	'anyChar'

Category	Regular Expression
Operator	= | || | && | == | != | < | <= | > | >= | + | - | * | / |! | [|]
Separator	: | . | { | } | (|)
Comment	// (anyChar | whitespace)* eol

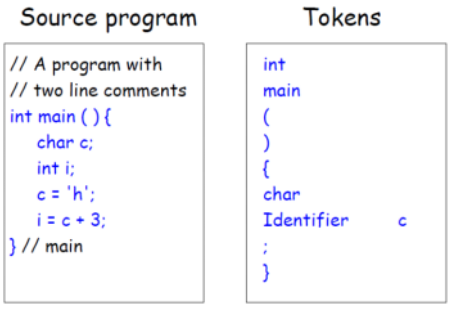
화면 캡처: 2022-04-07 오전 11:58

- **IDENTIFIERS**
  - a,b,c,sum,.. ID(모든 종류의 식별지)
- **LITERALS**
  - 123,'x',true,false
- **KEYWORDS**
  - Int , float, double, ..
- **OPERATORS**
  - +-*/-
- **PUNCTUATION**
  - ; , () {} ..

추가로 Lexier가 하는 역할

Lexical Analyzer는 parsing 하는데 필요 없는 것들은 버립니다. (예) whitespace, comment

1. Witespace 제거
2. Comment제거
3. Line counting
   a. 몇번째 라인인지 계산한다 line comment 처리하기 위해 별도로 구분 처리

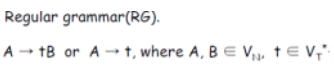


화면 캡처: 2022-04-07 오전 11:45

Regular languages

정의 방법

1. **Regular grammar**



화면 캡처: 2022-04-07 오전 11:48

t는 termianl A,B는 non terminal symbol이다

EX 1) 010으로 시작하는 모든 binary string 생성하는 문법
S-> 010A
A->0A | 1A | 입실론

이때 010은 Terminal symbol이고 A는 non terminal symbol이다

EX 2)

1. 영문자 도는 '_'로 시작하되 영문자와 숫자로 구성되는 변수이름
2. 모든 십진수 정수
3. JAVA 의 연산자 집합

모두 regular grammar를 이용하여 regular language로 정의할수있다

4. 유한개의 string으로 이루어진 집합은 정규 언어이다
   i. YES
   ii. RG를 이용해서 만들수있는 언어 , 모든 알파벳 , 숫자는 정규언어니까

2. **Regular expression**

Regular Expression	Language	
ε	{ε}	
a, where a∈T	{a}	
P|Q	$L_P \cup L_Q$	Union
PQ	$L_P L_Q$	Concatenation
P*	L_P^*	Closure

화면 캡처: 2022-04-07 오전 11:54

Regurlar Expression을 결합해서 새로운 RE 정의 할수있다

EX)
011으로 시작하는 바이너리 스트링의 집합 011(0|1)*
011으로 끝나는 바이너리 스트링의 집합 (0|1)*011

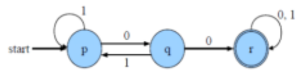
EX 2)
1.01 = {01}
2.0|1 = {0,1}
3.5*={5}*
4.(0|1)*={0,1}*

문헌에 따라 | 를 +로 표기하기도 한다

3. **Finite automata**

주어진 input string x에 대해 x ∈ L일때 YES라고 판정하는 기계 (turing machin과 비슷하다)

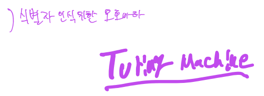
State Diagram

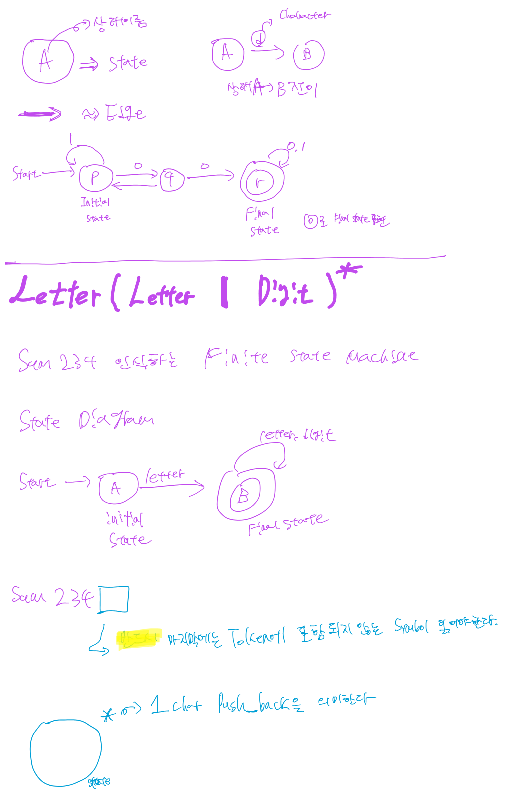


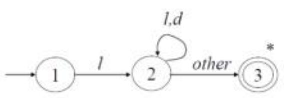
$(1+01)^*00(0+1)^*$

상태전이도

화면 캡처: 2022-04-07 오후 12:00







State diagram

입력 / 상태	letter	digit	other
1	2	error	error
2	2	2	3 (accept)

State transition table

화면 캡처: 2022-04-07 오후 12:18

Lexer 구현

모든 방식을 위의 구조로 만들면 복잡해진다
그래서 한 언어의 automata는 종류별로 token 인식하는 sub-automata들을 통합하여 구성할수있다

시작 상태에서 출발하여 token 한 개를 인식하고 다음 token을 인식하기 위해 다시 시작 상태로 돌아간다
이때 돌아가기전에 다음 토큰에 사용되는 글자를 push back하고 돌아가야함
->입력에서 토큰의 끝을 확인하기 위해서는 그 토큰 다음에 오는 글자를 하나 더 읽어야한다
이 글자는 다음 토큰의 일부이므로 다음 토큰의 인식을 위해 시작 상태로 돌아가기 전에 **읽지** 않은 상태로 되돌려 놓아야한다 pushback

Push back 구현 방법

1. Push back function 구현
    i. C 에 ungetch()

```
char ch;

char getNextChar() {
    char c;
    if (buffer != NULL) {
        c = ch; ch =NULL;
        return c;
    }
    else
        return getch();
}

void pushback(char a) { ch = a; }
```

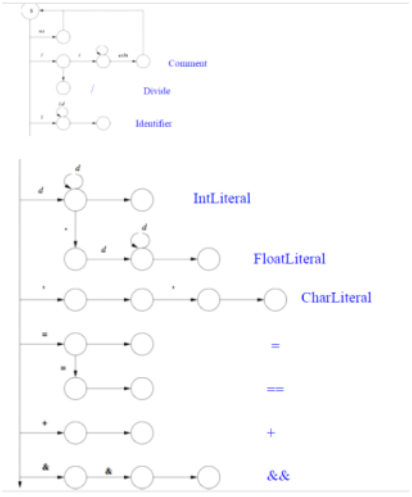
푸쉬백 함수의 구현(getNextChar())

화면 캡처: 2022-04-07 오후 12:23

2. 시작 상태로 돌아오기 전에 항상 한 글자 미리 읽어 두기

시작상태에는 항상 한 글자가 버퍼에 미리 읽혀져 있다고 가정한다. 즉, 첫 글자는 버퍼에서 읽는다.
• 따라서, 모든 sub-automata에서 토큰을 인식한 후에는 항상 다음 글자를 버퍼에 읽어놓고 시작상태로 돌아간다.

Automata가 첫 토큰을 인식할때는 어떻게 할까?
-> 토큰 인식에 해가 되지 않는 아무 문자 삽입해서 읽어옴



화면 캡처: 2022-04-07 오후 12:24

3. Transition from one state to another
    1. 시작 상태인 경우
        i. 버퍼 ch에 저장된 입력과 일치하는 arc를 따라서 다음상태로 이동한다. 일치하는 arc가 없다면 error
    2. 시작 상태가 아닌 경우
        i. Read the next character into ch
        ii. If ch와 일치하는 arc가 있다면
            1) String에 글자를 추가하고 다음상태로 이동
        iii. Else if there is an unlabeled arc
            1) Read a char into ch and follow that arc
            2) Token인식 완료
        iv. Else
            1) Error

From Design to code

```
Public Token next()
{
    do
    {
        If(isLetter(ch)
        {
            String speeling = concat(letters+digits);
            return Token.keyword(spelling);      //identifier, keyword 같이 받아서 마지막에 구분한다

        }
        Else if(isDigit(ch))
        {
            String number = concat(digits);
            If(ch!='.')
            {
                Return Token.mkIntLiteral(number);      //integer일때 반환
            }
            Number+=concat(digits);
            Return Token.mkFloatLiteral(number);     //float형일때 반환

        }
```

```
        Else switch(ch)
        {
            Case ' ': case "₩T" : case : '₩r' case eolnCh:

                Ch = nextch();
                Break;
            .....
        }


    }while(true)

    Private bollen isLetter(char c)
    {
        Return ch>='a' && chM<="z" || ch>="A" && ch<="Z";
    }


    Private String concat(String set)
    {
        StringBuffer r = new StringBuffer("");
        Do
        {
            r.append(ch);
            Ch = nextChar();
        }while(set.indexOf(ch)>=0);

        Return r.toString();
    }
}
```

6주차 과제 토론+ 프로그램 부연 설명

https://it-qa.com/how-are-lexical-errors-identified/

과제 목표 : clite의 lexical analyzer C로 구현

토론

1. **언어의 token을 regular expression으로 명세하라**

Category	Regular Expression
Keyword	bool | char | else | false | float | if | int | main | true | while
Identifier	letter (letter | digit)*
integerLit	digit+
floatLit	digit *.digit *
charLit	'anyChar'

Category	Regular Expression
Operator	= | || | && | == | != | < | <= | > | >= | + | - | * | / |! | [|]
Separator	: | . | { | } | (|)
Comment	// (anyChar | whitespace)* eol

2. **Token을 인식하는 Finite State Machine을 구현하라**

3. **Source program에 오류가 있을 경우 어떻게 하는 것이 좋을까 생각해보라.**

1. **Lexical level과 Syntax level의 오류는 어떤 차이가 있는가?**

예시

Int 2sb;
Int 2;
Lexical error는 compiler가 적절한 token의 sequence of character인가를 인지 못할때 발생
Valid token 만들수없을때
2ab는 변수 규칙에 맞지 않는다

Syntax error는 int 2처럼 문법이 맞지 않을때 발생
Token이 해당 문법에 맞지 않을때 발생

1. **Compiler가 만나는 오류의 종류를 level 별로 분류할 수 있는가? 그 종류를 나열해보라.**

a. **Syntax error-> 문법 오류**
b. **Sementic error-> 프로그램은 돌아가는데 의도한데로 나오지 않는 오류**
    i. **논리적으로는 맞는데 변수를 다른것을 썼다거나등의 이유로 원하는 결과가 나오지 않는점에서 논리적 오류와 차이가 있다**
c. **Lexical error**

2. **Lexer는 syntax level의 오류를 어떻게 처리하는 것이 좋을 지 생각해보라.**
    a. **Parser에서 얻은 문자열을 token으로 바꾸어서 syntax error 처리 한다**
    b. **Lexical은 문자가 잘못 되었거나 이런건 detect할수있지만 문법적으로 틀린것인지는 탐지 할수없느느데 그래서 sourece를 token으로 바꾸어서 parse tree만들어서 syntatical error를 탐지한다?**

Parser의 목적

1. 모든 syntax error를 발견하고 적절한 진단 메시지를 제공하여 error 해결하는것
2. 프로그램에 parse tree를 제공하는것

Lexical Error (Scanner error)

https://stackoverflow.com/questions/3484689/what-is-an-example-of-a-lexical-error-and-is-it-possible-that-a-language-has-no

character의 순서가 Token pattern과 맞지 않을때 발생한다
Token pattern과 일치하는지 본다

A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

출처: <https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm>

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

1. Signed integer는 범위 안에 있어야하는데 범위 밖에 숫자를 쓸수있을때?
2. printf("Geeksforgeeks");$ 과 같이 iligal character $가 있는 경우

A **syntax error** occurs when you write a statement that is not valid according to the grammar of the C++ language. This includes errors such as missing semicolons, using undeclared variables, mismatched parentheses or braces, etc...

Syntax analysis Top down parsing

2022년 4월 13일 수요일 오후 12:13

Syntactic Analysis(parsing)

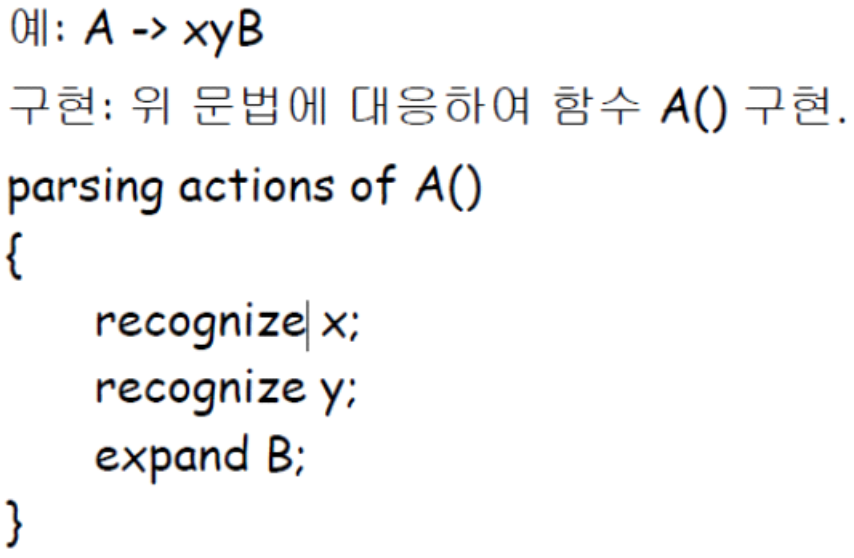
Recursive Descent parser

목적: 문장 구조 분석
Input : stream of tokens
Output: parse tree or AST
Algorithm : Recursice descent parsing

Recursive descent parsing

- Top down parsing
- Root node로 부터 내려가며 재귀적으로 left parse로 트리 생성
- Nonterminal을 규칙의 우변으로 확장
- 모든 leaf 노드가 terminal될때까지 반복

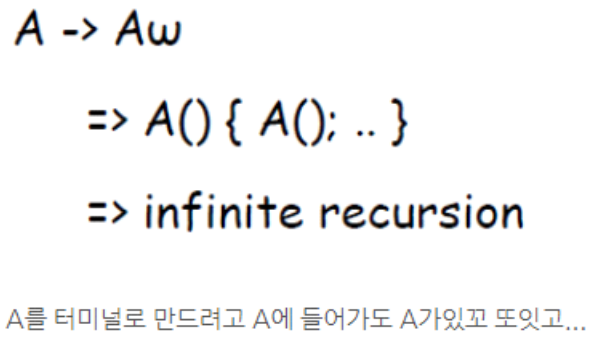
Non Terminal 을 확장하려면 대응하는 함수를 call해야한다



화면 캡처: 2022-04-13 오후 12:19

x,y는 terminal A는 non terminal이니까 확장 expand 하려면 함수를 call해야함

Left recurive rule이 있다면 Top down parsing 불가능

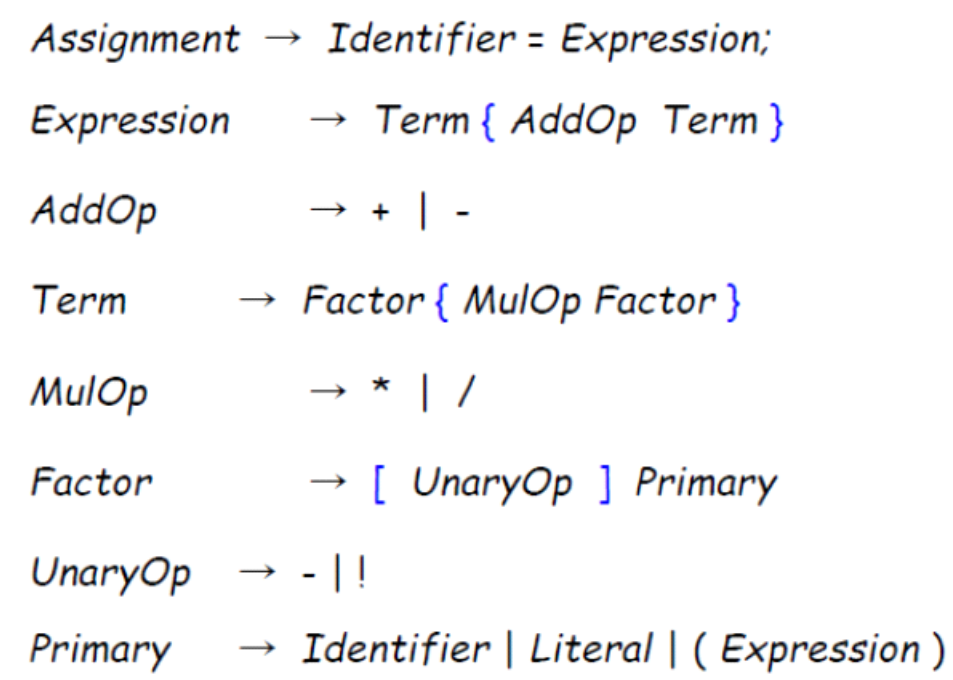


A를 터미널로 만드려고 A에 들어가도 A가있고 또있고...

화면 캡처: 2022-04-13 오후 12:20

무한 반복된다

- Clite BNF는 left- recursion을 포함하고 있다 -> recursive descent parsing 사용불가
- Clite의 BNF를 EBNF로 바꾸면 사용가능
    - EBNF는 BNF의 recursion을 iterator로 바꾼것이기 때문이다
    - Left recursion이 사라진 형태



화면 캡처: 2022-04-13 오후 12:22

Token is Terminal !!

Assignment -> Identifier = Expression ;

```
Assignment Node assignment( )
{
      //identfifier,Assign은 token이다
      match(Token.Identifier);
      match(Token.Assign);

      //expression은 non terminal이니까 함수 call
      expression( );
      match(Token.Semicolon);
      // make and return an Assignment_Node
}
```

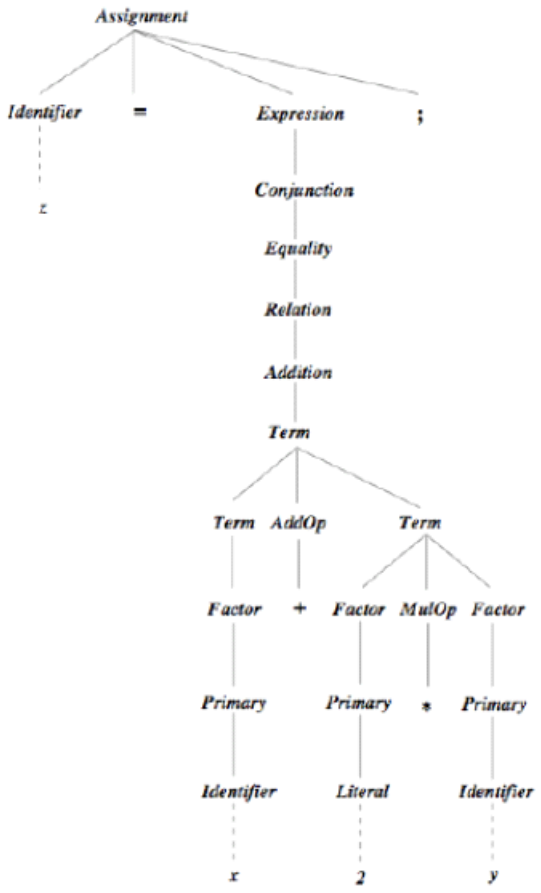
Expression -> Term {AddOp Term}

```
Expression Node expression()
{
      term( ); // make a Term subtree
      // assume that the next token has been read.
      while (isAddOp ())
```

```
            //..save the operator token
            token =lexer.next ( ); // get next token
            term( );
        }
        //..
        make and return an Expression_Node
}
```

표현식의 EBNF의 {}이거 어떻게 구현?
Lookahead를 한다 -> 0번이상 반복하는지 확인하는 방법

Parse Tree for z = x+2*y;



full parse tree

화면 캡처: 2022-04-13 오후 12:27

Rule 선택

A에 대한 생성규칙이 여러 개일 경우 어떤 규칙을 사용?

- Current token에 대응되는 rule 선택

그럼 A --> X | Y | ε 일때는?
First(X)함수를 적용: 일반적인 리컬시브 디센트 파서에서만 쓰임
-x가 생성하는 스트링을 시작할 수 있는 terminal들을 반환
First(X) = { a ∈T | X ∈ * a ω , ω ∈ (V N U V T)*

토큰이 X가 생성하는 string에서 시작하는 터미널이라면 A-> X적용
토큰이 Y가 생성하는 스트링을 시작하는 터미널이라면 A-> Y적용
아니면 A-> ε

Abstract Syntax Tree (AST)

- 간소화된 parse tree
- 의미적 필수 요소만 포함
- 의미는 같은데 각 언어의 다른 문법때문에 parse tree쓰는거 대신 AST쓰는게 더 간편해서 사용?

AST 생성하기

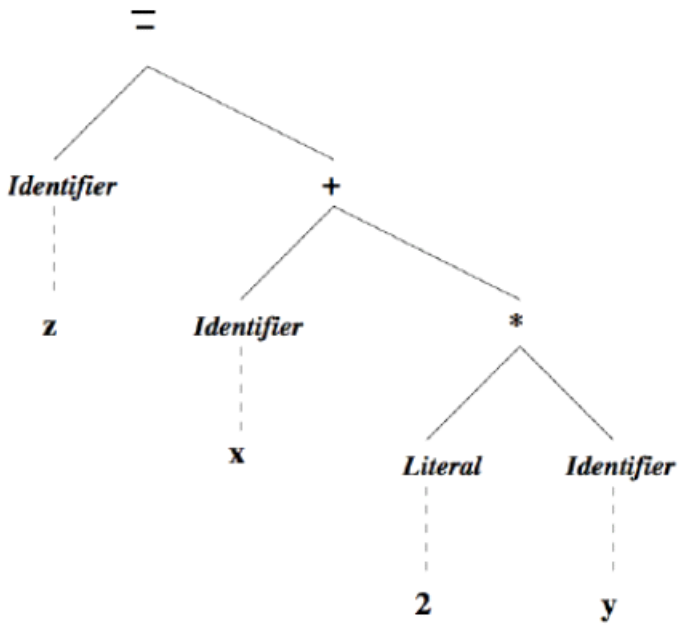
방법 1.

- Parsing 과정에서 직접 AST생성

방법 2.

- Syntax tree를 AST로 변환
  - Parse tree에 internal non terminal 삭제
  - Separator와 puctuation 터미널 심볼 제거
  - 모든 사소한 non terminal 제거
  - 남아 있는 non terminal을 leaf terminal로 대체

Parse tree-> AST결과가 항상 같은건 아님 사람마다 다를수있다



화면 캡처: 2022-04-13 오후 12:35

Abastract syntax 정의

의미중심의 문장 구조 정의

1. 프로그램은 선언부와 실행부로 구성
   EBNF: 프로그램-> int main(){Declarations Statements}
   Abs Syntax: Program = Declarations d ; Statements body

2. assignment 는 target 변수(x)와 source expression(y+2) 으로 이루진다

Assignment = VariableRef target; Expression source

Clite AST

Program = Declarations decpart ; Statements body
Declarations = Declaration*
Declaration = VariableDecl | ArrayDecl
VariableDecl = Variable v ; Type t
ArrayDecl = Variable v ; Type t ; Integer size
Type = int | bool | float | char

Statements = Statement*
Statement = Skip | Block | Conditional | Loop | Assignment
Skip =
Block = Statements
Conditional = Expression test ; Statement thenbranch , elsebranch
Loop = Expression test; Statement body
Assignment = VariableRef target ; Expression source
Expression = VariableRef | Value | Binary | Unary
VariableRef = Variable | ArrayRef
Variable = String id
ArrayRef = String id ; Expression index

Value = IntValue | BoolValue | FloatValue | CharValue
Binary = Operator op; Expression term1, term2
Unary = UnaryOp op; Expression term
Operator = BooleanOp | RelationalOp | ArithmeticOp
BooleanOp = && | ||
RelationalOp = == | != | < | <= | > |
ArithmeticOp = + | --| * | / |
UnaryOp = ! | --| float() | int()
IntValue = Integer intValue
FloatValue = Floate floatValue
BoolValue = Boolean boolValue
CharValue = Character charValue

AST 구현을 위한 class 정의 방법

- AST의 각 non terminal을 class정의
- 각 non terminal x는
  - X가 정의의 우변에 있으면 x를 y의 서브 클래스로 선언
  - X의 abstarcact syntax우변에 데이터 필드가 있으면 x를 클래스로 정의
  - 아닌경우는 x를 abstract class정의

강의 자료에 class로 구현한 AST 참고해라