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

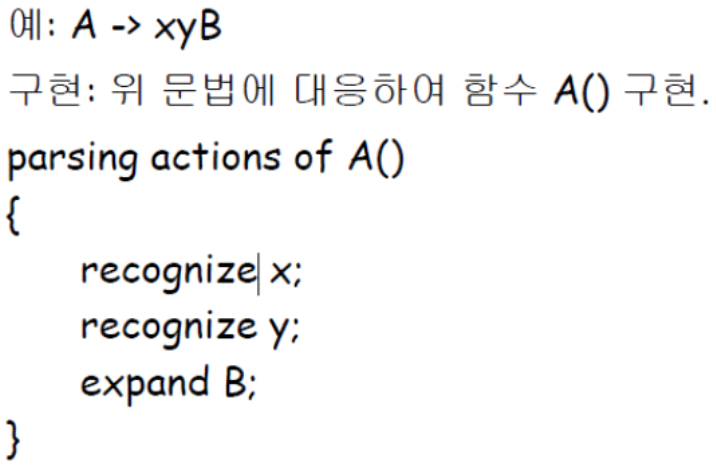
Output: parse tree or AST

Algorithm : Recursice descent parsing

Recursive descent parsing

- Top down parsing
- Root node로 부터 내려가며 재귀적으로 left parse로 트리 생성
- Non - terminal을 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

A -> Aω

=> A() { A(); .. }

=> infinite recursion

A를 터미널로 만드려고 A에 들어가도 A가있꼬 또잇고...

화면 캡처: 2022-04-13 오후 12:20

무한 반복된다

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

Assignment → *Identifier* = *Expression*;

Expression → *Term* { *AddOp* *Term* }

AddOp → + | -

Term → *Factor* { *MulOp* *Factor* }

MulOp → * | /

Factor → [*UnaryOp*] *Primary*

UnaryOp → - | !

Primary → *Identifier* | *Literal* | (*Expression*)

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()이 call되면

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

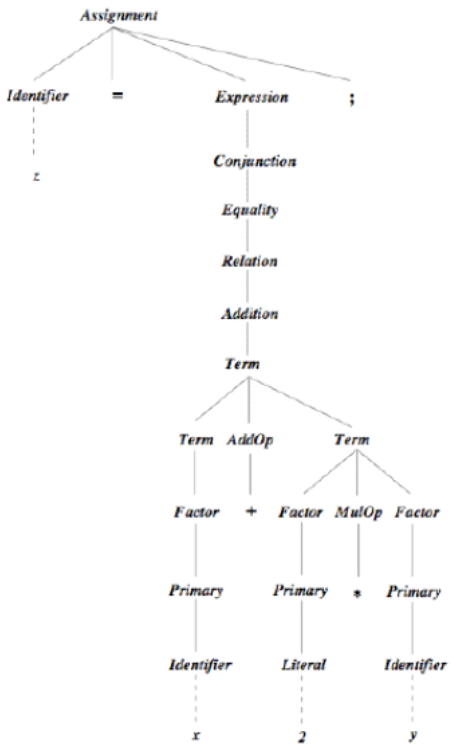
ex) Term1 + Term2 - Term3

term() -> Term1
isAddop() - > + {}은 ebnf에서 0번 이상 반복하는건데 이걸 인식하기 위해 사용 즉 연산자가 있는지 확인한다
lexer.next()을 통해 연산자를 저장한다
term() -> Term2

그리고 while(isAddop()) 조건으로 와서 (-)을 인식하면서 들어간다

Syntax 분석 해서 parse tree 구성하는 방법을 배우고 있다

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

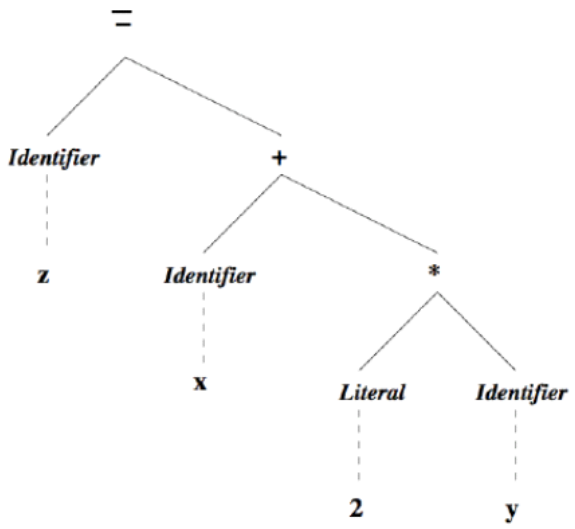
방법 2.

- 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

흔히 방법 1로 AST를 생성한다

Parse tree-> AST결과가 항상 같은건 아님 사람마다 다를수있다

밑에는 방법 2로 AST생성한 것이다



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

-> program은 선언부 d와 실행문 body로 이루어져 있다

Int main() {}이런것들은 syntax 구분할때는 필요하지만 의미적으로 보았을때 크게 필요하지 않다
의미적으로 필요한 부분만 추려서 나타낸다

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

Assignment = VariableRef target; Expression source

Abstract syntax of Clite

Program = Declaration2 decpart ; Statement2 body

Declaration2 = Declaration*

Declaration = VariableDecl | ArrayDecl

VariableDecl = Variable v ; Type t

ArrayDecl = Variable v ; Type t ; Integer size

Type = int | bool | float | char

Statement2 = Statement*

Statement = Skip | Block | Conditional | Loop | Assignment

Skip = //null

Block = Statements

Conditional = Expression test ; Statement thenbranch ; Statement 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 = + | --| * | / |

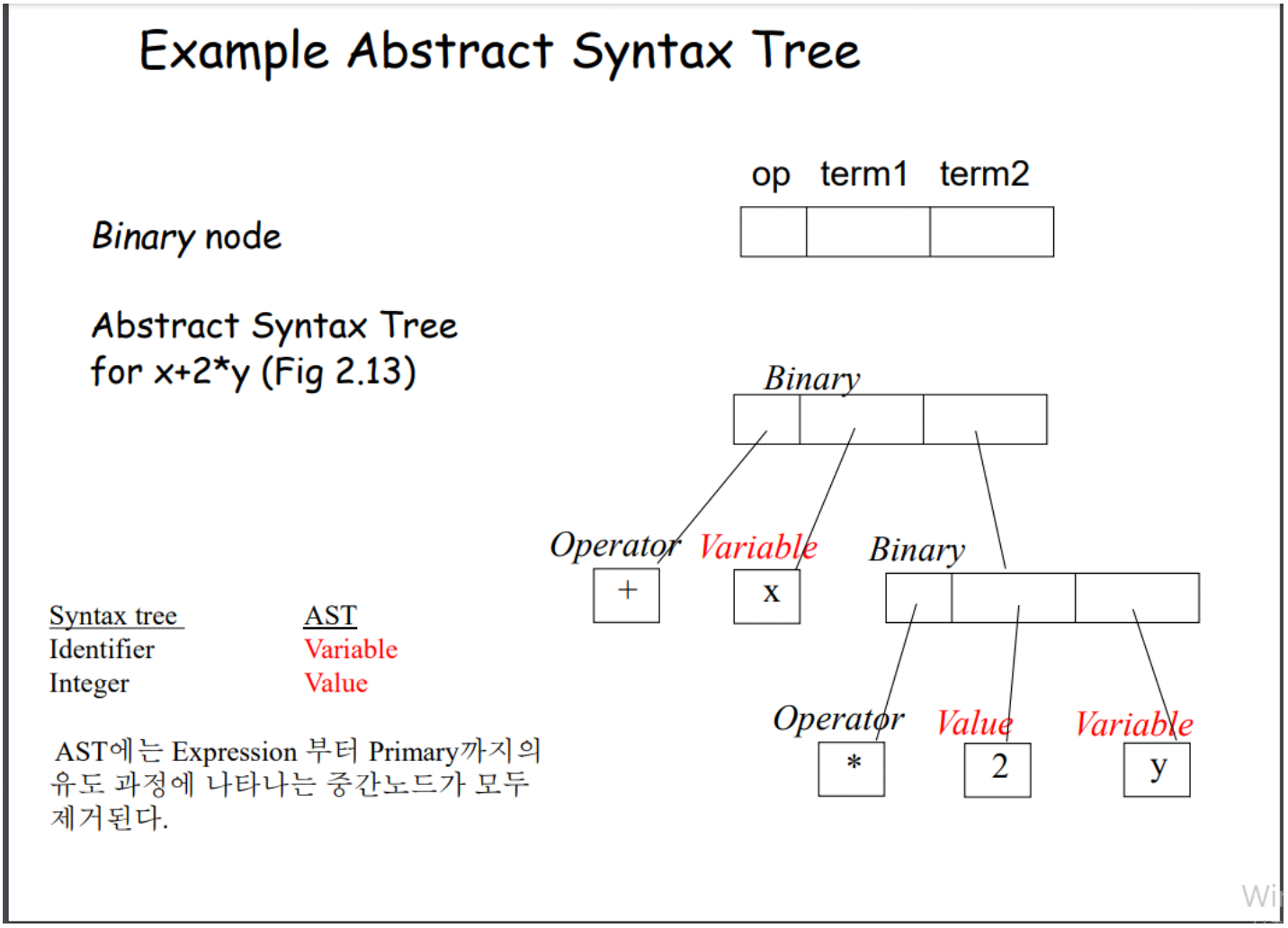
UnaryOp = ! | --| float() | int()

IntValue = Integer intValue

FloatValue = Floate floatValue

BoolValue = Boolean boolValue

CharValue = Character charValue



화면 캡처: 2022-04-19 오후 10:13

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

예) *VariableRef* = *Variable* | *ArrayRef*

```
abstract class VariableRef extends Expression { }
```

예) *Variable* = *String* id

```
class Variable extends VariableRef { String id }
```

Class를 이용한 Abstract Syntax 구현

```
abstract class Expression { }

abstract class VariableRef extends Expression { }
class Variable extends VariableRef { String id; }
class ArrayRef extends VariableRef { String id; Expression index}
class Value extends Expression { … }
class Binary extends Expression {
    Operator op;
    Expression term1, term2;
}
class Unary extends Expression {
    UnaryOp op;
    Expression term;
}
```

화면 캡처: 2022-04-19 오후 10:14

Parsing for Assignment

```
Assignment void assignment( ) {
  // Assignment → Identifier = Expression ;
  //   Assignment = Variable target; Expression source

  // create target node using Identifier
  Variable target = new Variable(match(Token.Identifier));
  match(Token.Assign);
 // create source node using Expression
  Expression source = expression( );
  match(Token.Semicolon);
 // create and return an Assignment node.
 return new Assignment(target, source);
}
```

화면 캡처: 2022-04-19 오후 10:14

```
  // match()

  private String match (TokenType t) {
      // check if the current token matches t
      String value = token.value();
      if (token.type().equals(t))
          token = lexer.next();         // get next token ready
      else
          error(t);
     return value;
  }
```

화면 캡처: 2022-04-19 오후 10:14

```
// error()

private void error(TokenType tok) {
    System.err.println(
        "Syntax error: expecting"
        + tok + "; saw: "  + token);
    System.exit(1);
}
```

화면 캡처: 2022-04-19 오후 10:14

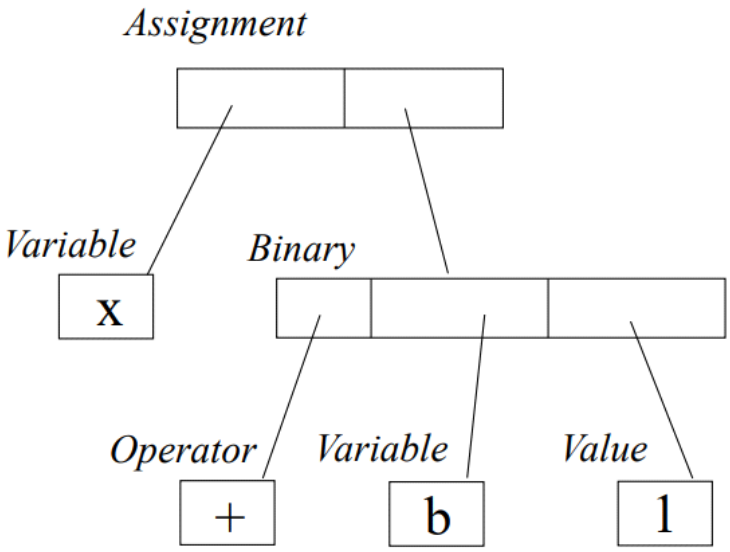
```
Expression expression( ) {
  // Expression → Term { AddOp Term }
  Expression e = term( );
  while (isAddOp()) {
    token = lexer.next( );
    Expression e2 = term( );
    e = new Binary(Token.AddOp, e, e2);
  }
    return e;
}

Expression term() {
  ...
} // 이하 교재 내용 참고
```

화면 캡처: 2022-04-19 오후 10:15

Example Abstract Syntax Tree

x = b + 1



화면 캡처: 2022-04-19 오후 10:15

---

pc2.c parse tree