3장 언어 설계와 파서 구현

3.1 프로그래밍 언어 S
3.2 추상 구문 트리
3.3 어휘 분석기 구현
3.4 파서 구현

3.1 프로그래밍 언어 S

샘플 프로그래밍 언어의 주요 설계 목표

(1) 간단한 교육용 언어로 쉽게 이해하고 구현할 수 있도록 설계한다.

(2) 대화형 인터프리터 방식으로도 동작할 수 있도록 설계한다.

(3) 프로그래밍 언어의 주요 개념을 쉽게 이해할 수 있도록 설계한다.

수식, 실행 문장, 변수 선언, 함수 정의, 예외 처리, 타입 검사 등

(4) 블록 중첩을 허용하는 블록 구조 언어를 설계한다.

전역 변수, 지역 변수, 유효범위 등의 개념을 포함.

샘플 프로그래밍 언어의 주요 설계 목표

(5) 실행 전에 타입 검사를 수행하는 강한 타입 언어로 설계한다.

안전한 타입 시스템을 설계하고 이를 바탕으로 타입 검사기를 구현.

(6) 주요 기능을 점차적으로 추가하면서 이 언어의

어휘분석기, 파서, AST, 타입 검사기, 인터프리터 등을 순차적으로 구현.

[언어 S의 문법]

<program> → {<command>}

<command> → <decl> | <stmt> | <function>

<decl> → <type> id [=<expr>];

<stmt> → id = <expr>;

| '{' <stmts> '}'

| if (<expr>) then <stmt> [else <stmt>]

| while (<expr>) <stmt>

| read id;

| print <expr>;

| let <decls> in <stmts> end; 선언하고 in 에서 사용 하고 end

<stmts> → {<stmt>}

<decls> → {<decl>}

<type> → int | bool | string

프로그래밍 언어 S

- 언어 S의 프로그램
  - 명령어(<command>)들로 구성된다.
- 명령어
  - 변수 선언(<decl>)
  - 함수 정의(<function>)
  - 실행 문장(<stmt>)

프로그래밍 언어 S

- 실행 문장
  - 대입문, 조건 if 문, 반복 while 문,
  - 입력 read 문, 출력 print 문
  - 복합문 : 괄호로 둘러싸인 일련의 실행 문장들
  - let 문: 지역 변수를 선언과 일련의 실행 문장들
- 변수 선언
  - 변수 타입은 정수(int), 부울(bool), 스트링(string)

예제 프로그램

[예제 1]

```
>> print “hello world!”;
hello world!
>> int x = -5;
>> print x;
-5
>> x = x+1;
>> print x*x;
16
>> if (x>0)
then print x; else print -x;
4
```

프로그램 S
이 정도면 돌아갈 수 있다고, 처리 가능함.
Why? ⇒ 앞에서 선언 다 해줬기 때문.

예제 프로그램

[예제 2]

```
let int x = 0; in
    x = x + 2;
    print x;
end;
```

[예제 3]

```
let int x; int y; in
    read x;
    if (x>0) then
        y = x;
    else y = -x;
    print y;
end;
```

예제 프로그램

[예제 4]

```
let int x=0; int y=1; in
    read x;
    while (x>0) {
        y = y * x;
        x = x-1;
    }
    print y;
end;
```

예제 프로그램

[예제 5]

```
>> fun int square(int x) return x*x;
>> print square(5);
25
```

[예제 6]

```
>> fun int fact(x)
     if (x==0) then return 1;
     else x*fact(x-1);
>> print fact(5);
120
```

— 반복. 재귀함수

3.2 추상 구문 트리

파서와 AST

- 어휘 분석기(lexical analyzer)
  - 입력 스트링을 읽어서 토큰 형태로 분리하여 반환한다.
- 파서(parser)
  - 입력 스트링을 (재귀 하강) 파싱한다.
  - 해당 입력의 AST를 생성하여 반환한다
- 추상 구문 트리(abstract syntax tree, AST)
  - 입력 스트링의 구문 구조를 추상적으로 보여주는 트리
- 인터프리터(Interpreter)
  - 각 문장의 AST를 순회하면서 각 문장의 의미에 따라 해석하여 수행한다.

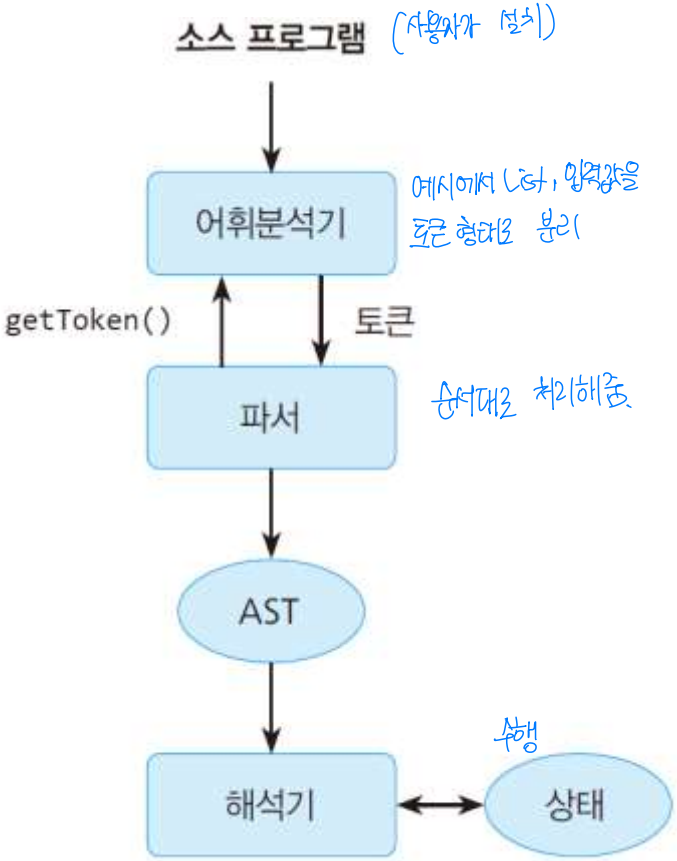


그림 3.1 어휘분석기, 파서, 해석기 구현

유도 트리

- [수식 문법 1] EBNF

  <expr> → <term> {+ <term>}

  <term> → <factor> {* <factor>}

  <factor> → [-] (number | id | '('<expr>')') 둘 다 쓸 수 없음

  쓸수 없거나 없거나

- a + b * c의 유도 과정

  <expr>

  => <term> + <term>

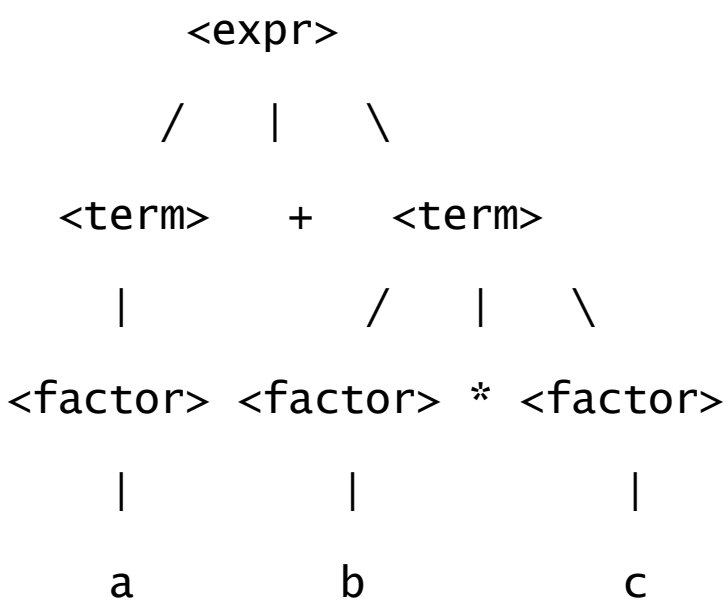
  => <factor> + <term>

  => <factor> + <factor> * <factor>

  => a + b * c

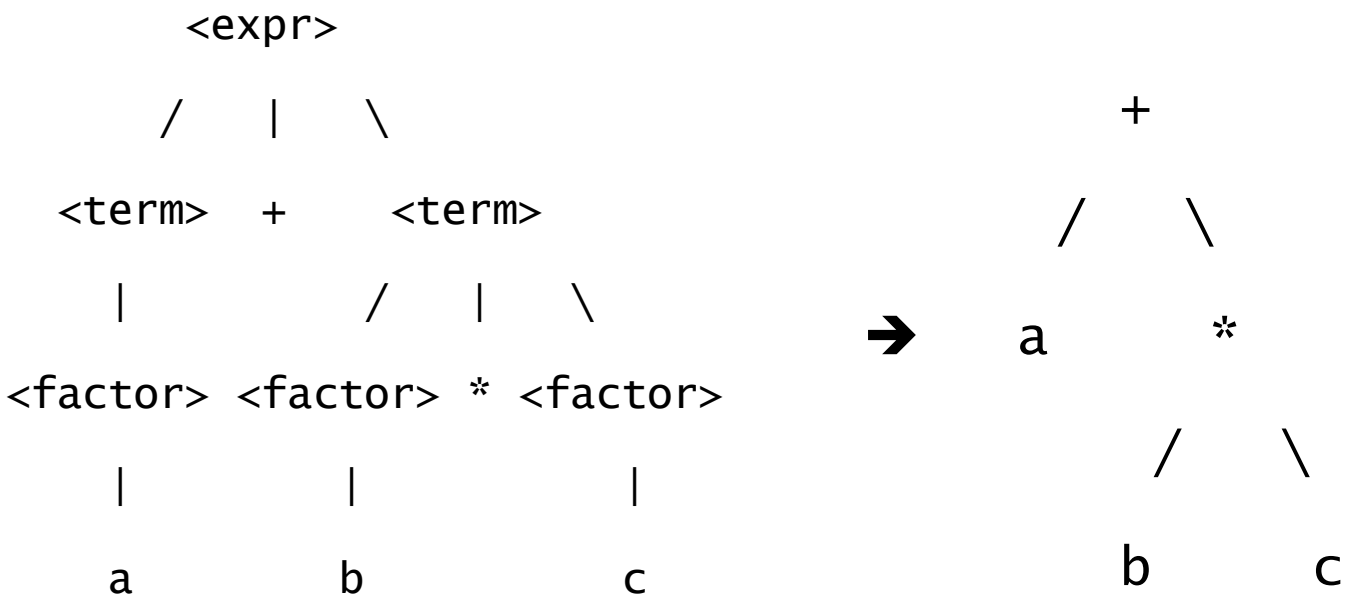
- 실제 파싱에서 a, b, c와 같은 변수 이름들은 모두 id로 처리되나 여기서는 이해를 위해 이름을 그대로 사용함

유도 트리



추상 구문 트리

- 추상 구문 트리(abstract syntax tree, AST)
  - 입력 스트링의 구문 구조를 추상적으로 보여주는 트리
  - 실제 유도 트리에 나타나는 세세한 정보를 모두 나타내지는 않음.
- 수식의 AST
  - 연산을 중심으로 요약해서 표현



수식 Expr의 AST

- 수식(Expr)의 AST
  - 이항연산 수식과 단항연산 수식으로 구분하여 구현

```
   Binary          Unary
  /  |  \          /   \
Expr Operator Expr     Operator Expr
```

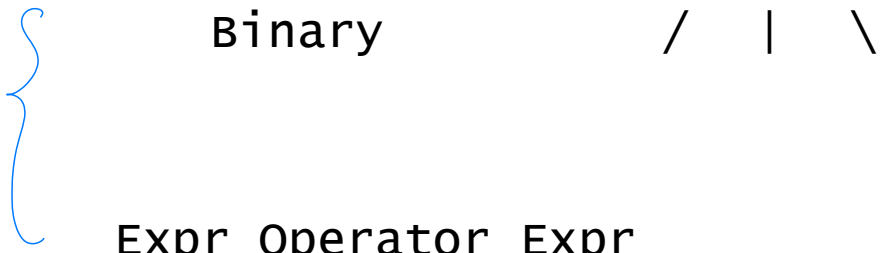
- 예 : a + b * c

```
     +                Binary
    /  \              /  |  \
   a    *     ==>     Id   +  Binary
       / \                  /  |  \
      b   c                Id  *   Id
```

수식의 AST 구현

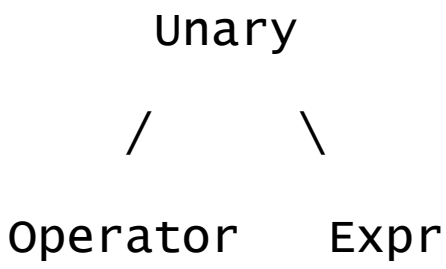
- 이항 연산(Binary) 수식의 AST 구현

```
class Binary extends Expr {
    // Binary = Operator op; Expression expr1, expr2
    Operator op;
    Expr expr1, expr2;
    Binary (Operator o, Expr e1, Expr e2) {
        op = o; expr1 = e1; expr2 = e2;
    } // binary
}
```

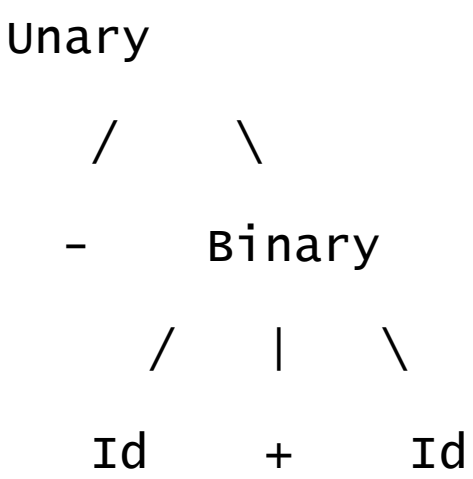


단항 연산 수식의 AST

- 단항 연산(Unary) 수식의 AST



- 예 : -(a+b)의 AST



수식(Expr)의 AST

- 어떤 것들이 수식이 될 수 있는가?
  - 이항 연산(Binary), 단항 연산(Unary)
  - 그 외의 수식은 없는가?
  - 식별자(Identifier), 값(Value)도 하나의 수식이 될 수 있다.

- 수식(Expr)의 AST 노드
  - Expr = Identifier | Value | Binary | Unary

➔ 수식은 변수, 값, 이항연산, 단항 연산이 될수 있다.

변수 선언의 AST

- 구문법

  <type> id = <expr> : 타입이름, 변수이름, 초기화 수식 으로 구성

  - ➔ 선언문 : 어떤 변수는 어떤 값으로 초기화

- AST

  - `Decl = Type type; Identifier id; Expr expr`

```
          Decl
        /  |  \
    Type  Id  Expr
```

- 예 `int x = 0;`

```
          Decl
        /  |  \
    Type  Id  Value
    int   x     0
```

대입문 Assignment의 AST

- 구문법 (대입문)
  - id = <expr>;

- AST 할당
  - Assignment = Identifier id; Expr expr

```
Assignment
   /    \
Id      Expr
```

```
class Assignment extends Stmt {
    Identifier id;
    Expr expr;
    Assignment (Identifier t, Expr e) {
        id = t;
        expr = e;
    }
}
```

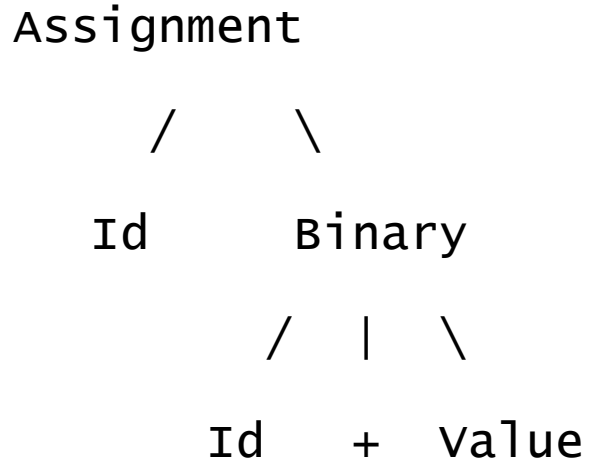
대입문 Assignment 문의 AST

- 예

x = x + 1;

x의주소

L x의 값 (real value)



Read 문, Print 문의 AST

- read, print 문의 구문법

  read id;　　print <expr>;

- read, print 문의 AST

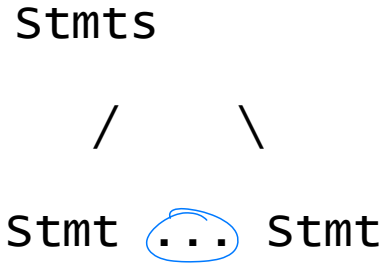
복합문의 AST

- 구문법 (반복문)

  <stmts> → {<stmt>}

- AST

  Stmts = Stmt*



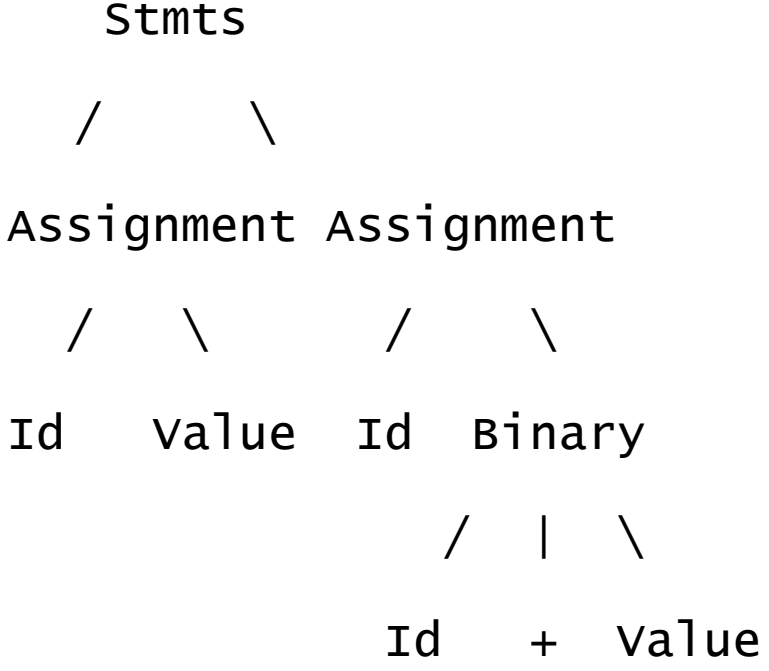

```
class Stmts extends Stmt {
    public ArrayList<Stmt> stmts =
    new ArrayList<Stmt>( );
}
```

복합문의 AST

- 예

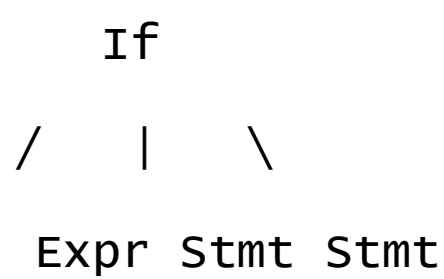
```
{
   x = 0;
   x = x + 1;
}
```

// 할당 } 2개 이상 처리하는 것 → 복합문



If 문의 AST

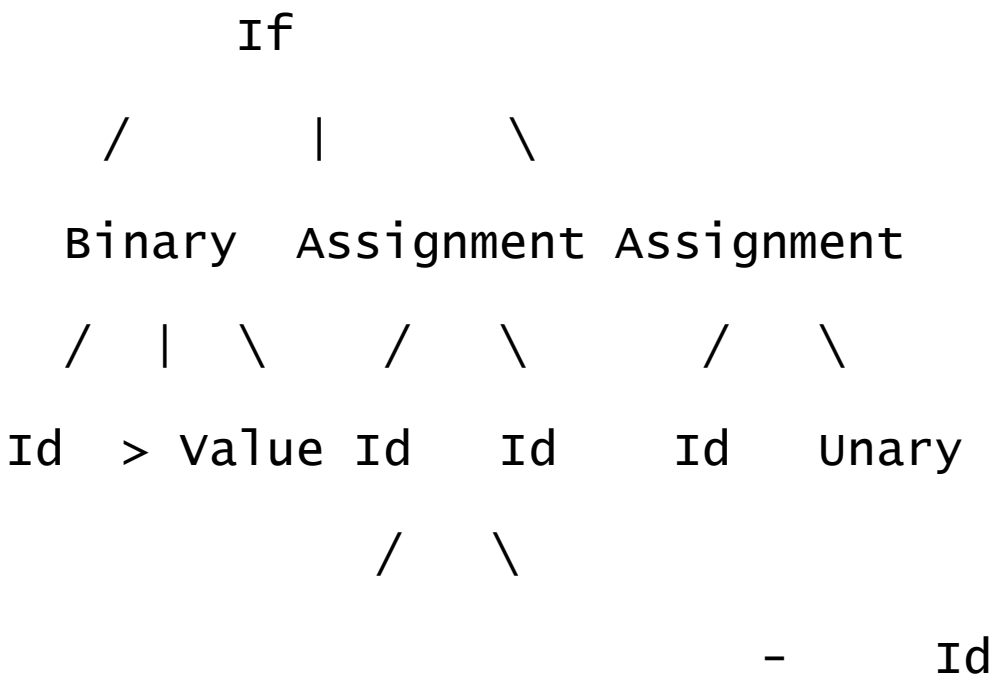
- 구문법
  - if (<expr>) then <stmt> [else <stmt>]
- AST
  - If = Expr expr; Stmt stmt1; Stmt stmt2



If 문의 AST 예

- 예

```
if (x > 0) then y = x;
else y = -x;
```



While 문의 AST

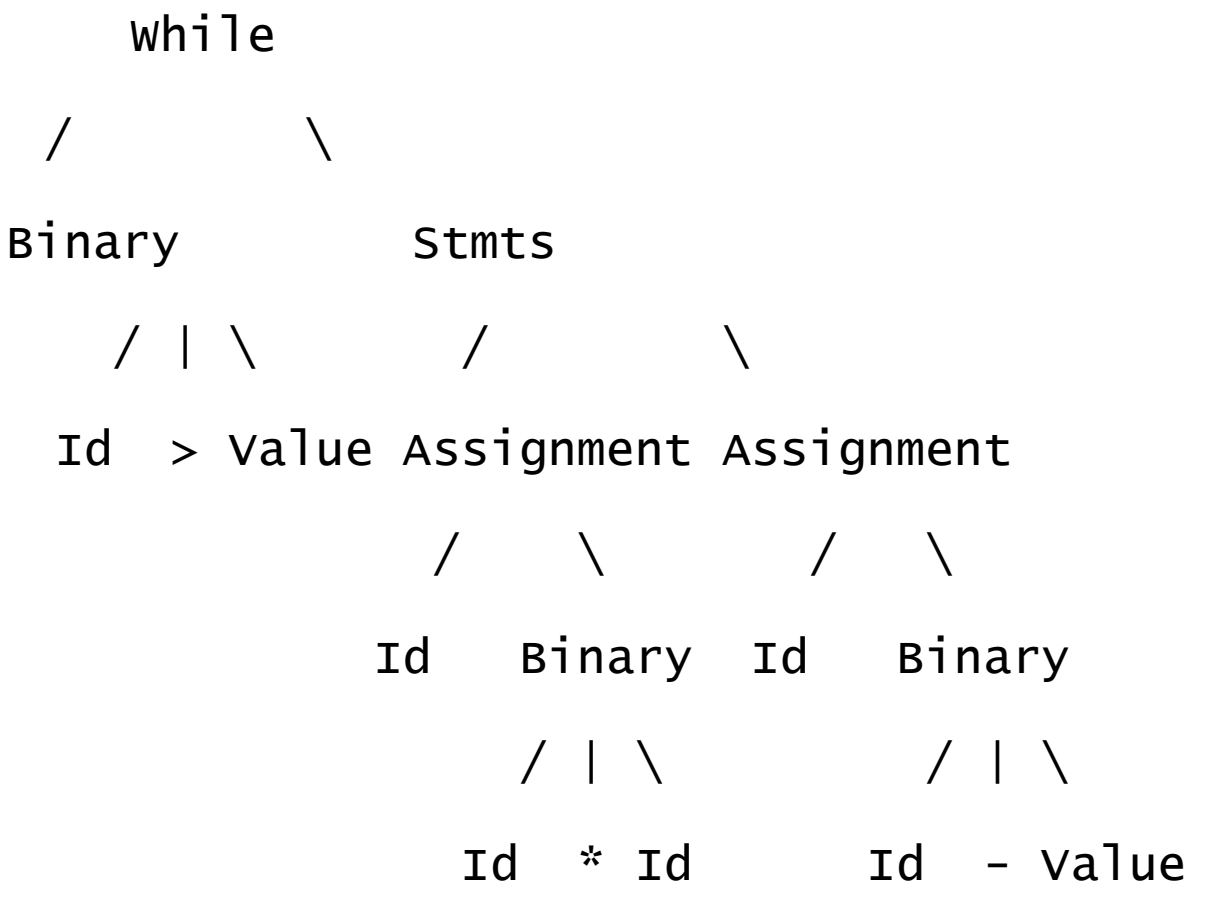
- 구문법
  - while '('<expr>')' <stmt>
- AST
  - While = Expr expr; Stmt stmt;

```
While
   /   \
Expr   Stmt
```

While 문의 AST 예

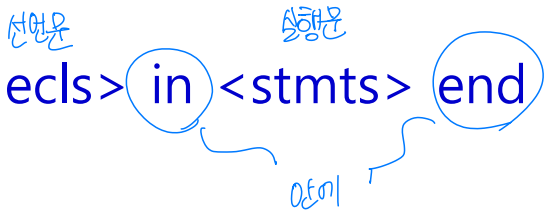
- 예

```
while (x > 0) {
    y = y * x;
    x = x - 1;
}
```



Let 문의 AST

- 구문법
  - let <decls> in <stmts> end



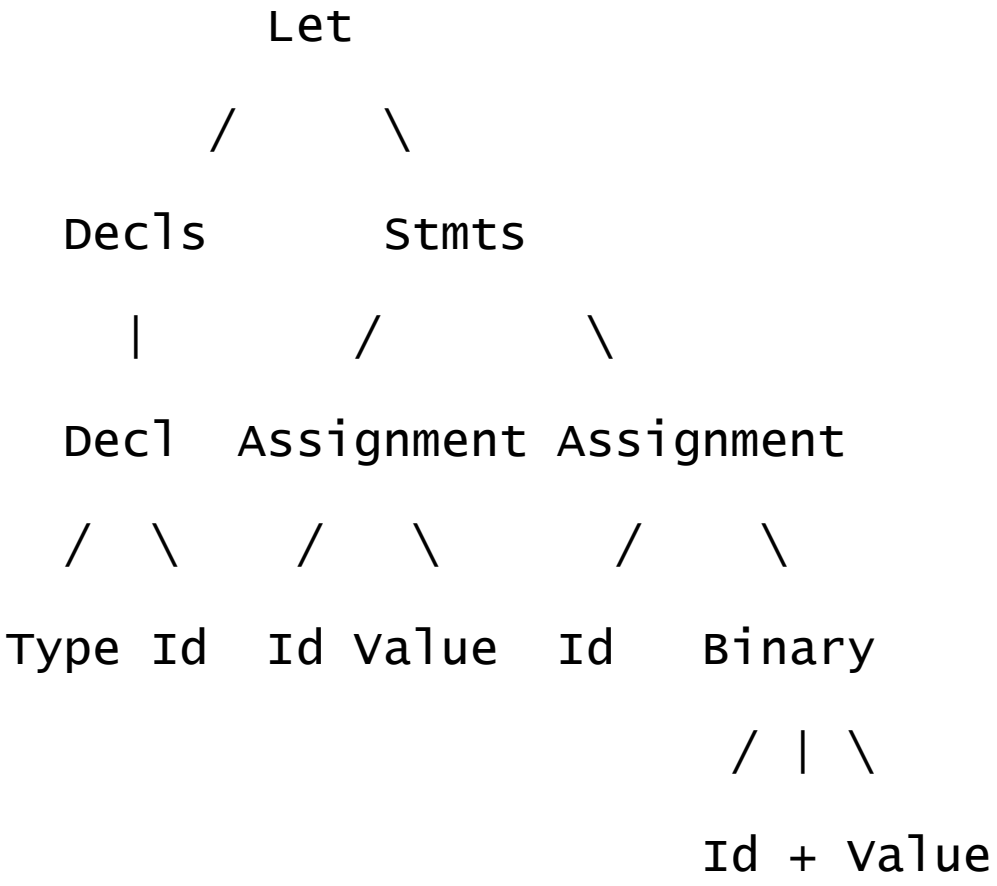
- AST
  - Let = Decls decls; Stmts stmts;

```
Let

  /   \

Decls  Stmts
```

Let 문의 AST 예

- 예

```
let int x; in
    x = 0;
    x = x + 1;
end;
```



3.3 어휘 분석기 구현

어휘분석과 토큰

- 어휘 분석(lexical analysis)
  - 소스 프로그램을 읽어 들여 토큰(token)으로 분리
  - 어휘 분석기(lexical analyzer) 또는 스캐너(scanner)

- 토큰
  - 문법에서 터미널 심볼에 해당하는 문법적 단위
  - 식별자(identifier)
  - 상수 리터럴(constant)
  - 예약어(keyword)
  - 연산자(operator)
  - 구분자(delimiter)

예약어

- 예약어 또는 키워드
  - 언어에서 미리 그 의미와 용법이 지정되어 사용되는 단어

- 언어 S의 예약어 이름과 해당 스트링
  - BOOL("bool"), TRUE("true"), FALSE("false"), IF("if"),
  - THEN("then"), ELSE("else"), INT("int"), STRING("string"),
  - WHILE("while"), VOID("void"), FUN("fun"), RETURN("return"),
  - LET("let"), IN("in"), END("end"), READ("read"), PRINT("print")

  - 실제 프로그램 개발에서 예약어와 내장함수는 변수나 함수명으로 사용하지 않는다. 예약어는 사용이 불가하고 내장함수 명은 사용 시 의미가 바뀐다.

식별자

- 식별자
  - 변수 혹은 함수의 이름을 나타내며 토큰 이름은 ID라고 하자.
  - 식별자는 첫 번째는 문자이고 이어서 0개 이상의 문자 혹은 숫자들로 이루어진 스트링

- 정규식(regular expression) 형태로 표현
  - ID = letter(letter | digit)*
  - letter = [a-zA-Z]
  - digit = [0-9]

정규식

[정의 1] M과 N이 정규식이면 다음은 모두 정규식이다.

(1) x　　문자 x를 나타낸다.

(2) M | N　　M 또는 N을 표현한다.

(3) MN　　M 다음에 N이 나타나는 접합을 표현한다.

(4) M*　　M이 0번 이상 반복됨을 표현한다.

추가적으로 다음과 같은 간단 표기법을 사용할 수 있다.

M^+　　MM*를 나타내며 M이 1번 이상 반복됨을 표현한다.

M?　　M이 0번 또는 1번 나타남을 표현한다.

[..]　　문자 집합을 나타낸다.

정규식

- 문자 집합 예
  - 모음 집합 [aeiou] = a | e | i | o | u
  - 대문자 집합 [A-Z]
  - 숫자 집합 [0-9]

- 예
  - letter = [a-zA-Z]
  - digit = [0-9]
  - 정수리터럴 NUMBER = digit^{+}
    하나 이상의 숫자들로 이루어진 스트링
  - 식별자 ID = letter(letter | digit)*

  - 생각해 보자. 정규식의 의미와 이메일을 정규식으로 표현하려면?

연산자/구분자

- 연산자
  - 언어 S에서 사용되는 연산자
  - ASSIGN("="), EQUAL("=="), LT("<"), LTEQ("<="), GT(">"), GTEQ(">="), NOT("!"), NOTEQ("!="), PLUS("+"), MINUS("-"), MULTIPLY("*"), DIVIDE("/"), AND("&"), OR("|")

- 구분자 — 코딩 시 혼연 헤깔릴생
  - 언어 S에서 사용되는 구분자
  - LBRACE("{", left brace 중괄호), RBRACE("}"),
  - LBRACKET("[", | left bracket 대괄호), RBRACKET("]"), LPAREN("(", left parentheses 왼쪽 소괄호), RPAREN(")"),
  - SEMICOLON(";"), COMMA(","), EOF("<<EOF>>")

토큰 구현

```
enum Token {  앞에 나온 편의 사전 정의
    BOOL("bool"), TRUE("true"), FALSE("false"), IF("if"),
    THEN("then"), ELSE("else"), INT("int"), STRING("string"),
    WHILE("while"), VOID("void"), FUN("fun"), RETURN("return"),
    LET("let"), IN("in"), END("end"), READ("read"), PRINT("print"),
    EOF("<<EOF>>"),
    LBRACE("{"), RBRACE("}"), LBRACKET("["), RBRACKET("]"),
    LPAREN("("), RPAREN(")"), SEMICOLON(";"), COMMA(","),
    ASSIGN("="), EQUAL("=="), LT("<"), LTEQ("<="), GT(">"),
    GTEQ(">="), NOT("!"), NOTEQ("!="), PLUS("+"), MINUS("-"),
    MULTIPLY("*"), DIVIDE("/"), AND("&"), OR("|"),
    ID(""), NUMBER(""), STRLITERAL("");
    private String value;
    private Token (String v) { value = v; }
    public String value( ) { return value; }
        …
```

어휘분석기 getToken 메소드

- getToken( ) 메소드
  - 호출될 때마다 다음 토큰(token)을 인식하여 리턴한다.

(1) 읽은 문자가 알파벳 문자: 식별자 아니면 예약어
  - 다음 문자가 알파벳 문자나 숫자인 한 계속해서 다음 문자를 읽는다.
  - 읽은 문자열이 식별자인지 예약어인지 구별하여 해당 토큰을 리턴한다.

(2) 읽은 문자가 숫자: 정수리터럴
  - 다음 문자가 숫자인 한 계속해서 읽어 정수리터럴을 인식하고 이를 나타내는 NUMBER 토큰을 리턴한다.

(3) 나머지는 읽은 문자에 따라 연산자, 구분자 등을 인식하여 리턴한다.

어휘 분석

- Lexer
  - 입력을 읽어서 호출될 때마다 하나의 토큰을 반환한다.
  - 키워드, 수, 변수 이름, 기타 문자 처리한다.

```
public class Lexer {
    . . .
    public Token getToken( ) {
        • 예약어(예 if)     return Token.IF;
            ...
        • 예약어(예 print) return Token.PRINT;
        • 정수             return Token.NUMBER.setValue(s);
        • 식별자           return Token.ID.setValue(s);
        • 연산자(예 +)     return Token.PLUS;
        • 구분자(예 ;)     return Token.SEMICOLON;
    }
}
```

3.4 파서 구현

파서의 구성

- 어휘 분석기(lexical analyzer)
  - 입력 스트링을 읽어서 토큰 형태로 분리하여 반환한다.
- 파서(parser)
  - 입력 스트링을 재귀 하강 파싱한다.
  - 해당 입력의 AST를 생성하여 반환한다
- 추상 구문 트리(abstract syntax tree, AST)
  - 입력 스트링의 구문 구조를 추상적으로 보여주는 트리

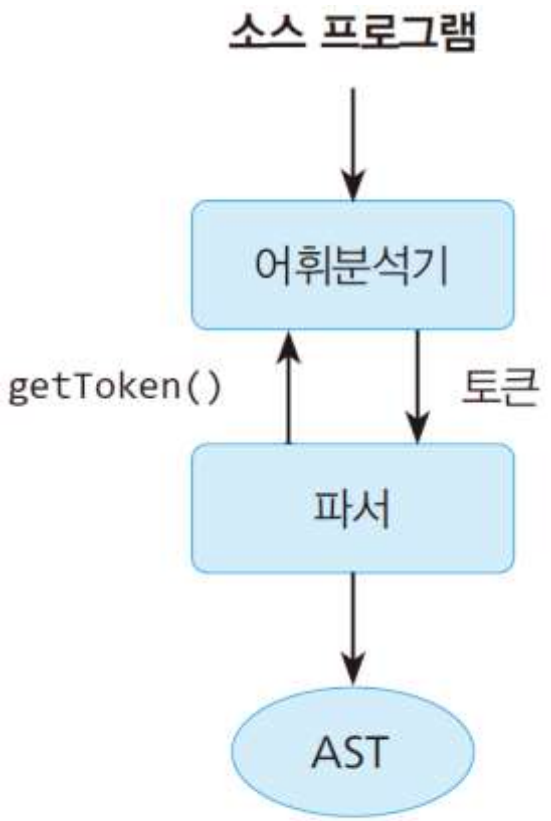
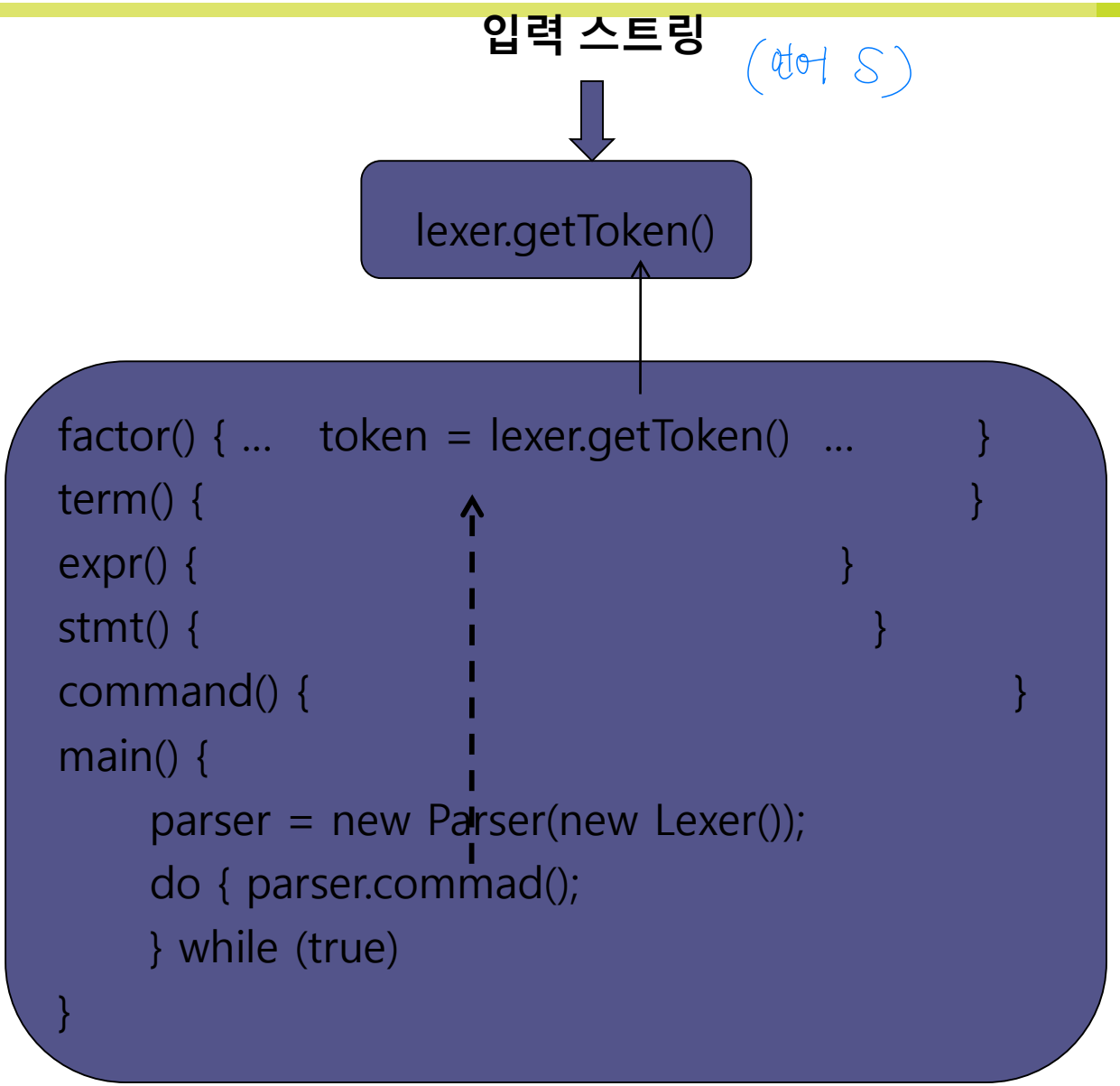


그림 3.2 파서의 구조

언어 S의 파서 구성

Lexer

Parser



파서

- 입력 스트링을 명령어 단위로 파싱하면서 AST를 생성하여 반환한다.

```
public class Parser {
    Token token;          // 다음 토큰 저장 변수
    Lexer lexer;

    public Parser(Lexer l) {
        lexer = l;
        token = lexer.getToken();      // 처음 토큰 읽기
    }
    public static void main( ) {  // <program> -> { <command> }
        parser = new Parser(new Lexer());
        System.out.print(">> ");
        do {
            Command command = parser.command();
            System.out.print("₩n>> ");
        } while(true);
    }
```

각 해당하는 토큰 값 변환

파서

- Command command( )
  - 명령어(변수 선언, 함수 정의, 문장)를 읽고 파싱하면서 해당 AST를 구성하여 반환한다.
  - <command> → <decl> | <stmt> | <function>
- Decl decl( )
  - 변수 선언을 읽고 파싱하면서 해당 AST를 구성하여 반환한다.
- Stmt stmt( )
  - 각 문장을 읽고 파싱하면서 해당 AST를 구성하여 반환한다.
- Expr expr( )
  - 수식을 읽고 파싱하면서 해당 AST를 구성하여 반환한다.

파서 구현: 명령어

```
public Command command( ) {
    // <command> -> <decl> | <function> | <stmt>
    if (isType()) {
        Decl d = decl( );           // 변수 선언 파싱
        return d;
    }
    if (token == Token.FUN) {
        Function f = function( ); // 함수 정의 파싱
        return f;
    }
    if (token != Token.EOF)) {
        Stmt s = stmt( );           // 실행 문장 파싱
        return s;
     }
    return null;
}
```

파서 구현: 변수 선언

- 구문법

  <decl> → <type> id [=<expr>];

  <type> → int | bool | string

```
private Decl decl () {
    Type t = type();          // 타입 이름 파싱
    String id = match(Token.ID);     // 변수 이름(식별자) 매치
    Decl d = null;
    if (token == Token.ASSIGN) {
        match(Token.ASSIGN);  // 대입 연산자 매치
        Expr e = expr( );          // 초기화 수식 파싱
        d = new Decl(id, t, e);  // 초기화 있는 AST 생성
     } else d = new Decl(id, t);  // 초기화 없는 AST 생성
     match(Token.SEMICOLON);// 세미콜론 매치
     return d;              // AST 리턴
}
```

Statement 파싱

```
Stmt stmt( ) {
    // <stmt> -> <assignment> | <ifStmt> | <whileStmt> | '{' <stmts> '}' | <letStmt> | ...
    Stmt s;
    switch (token) {
    case ID:            // 대입문 파싱: assignment
        s = assignment( );  return s;
    case LBRACE:
        match(Token.LBRACE); s = stmts( ); match(Token.RBRACE);
        return s;
    case IF:            // if 문 파싱: ifStmt
        s = ifStmt( );  return s;
    case WHILE:             // while 문 파싱: whileStmt
        s = whileStmt( ); return s;
    case LET:           // let 문 파싱: letStmt
        s = letStmt( ); return s;
    ...
    default: error("Illegal statement"); return null;
    }
}
```

대표문자 따로 처리

파서 구현: Assignment 문

- 구문법
  - <assignment> → id = <expr>;
- 파서 구현

```
Assignment assignment() {
    Identifier id = new Identifier(match(Token.ID));  // 식별자 매치
    match(Token.ASSIGN);     // 대입 기호 '=' 매치
    Expr e = expr( );        // 수식(expr) 파싱
    match(Token.SEMICOLON);  // 세미콜론
    return new Assignment(id, e); // AST 노드 생성하여 리턴
}
```

→ 이것만 있으면 파서 동작 하는건가? 노노

match 함수

- match( )
  - 현재 토큰을 매치하고 다음 토큰을 읽는다

```
private String match(Token t) {
    String value = token.value();
    if (token == t)
        token = lexer.getToken();
    else
        error(t);
    return value;
}
```

파서 구현: 복합문

- 구문법

  <stmts> → { <stmt> }

- 파서 구현

```
Stmts stmts ( ) {
    Stmts ss = new Stmts( ); //  빈 복합문 AST 생성
    while((token != Token.RBRACE) && (token != Token.END))
        ss.stmts.add(stmt( ));  // 문장 파싱하고 그 AST를 복합문 AST에 추가
    return ss;                  // 복합문 AST 리턴
}
```

파서 구현: If 문

- 구문법

  <ifStmt> → if (<expr>) then <stmt>
  [else <stmt>]

- 파서 구현

```
If ifStmt ( ) {
    match(Token.IF);
    match(Token.LPAREN);
    Expr e = expr( );
    match(Token.RPAREN);
    match(Token.THEN);
    Stmt s1 = stmt( );
    Stmt s2 = new Empty();
    if (token == Token.ELSE){
        match(Token.ELSE);
        s2 = stmt( );
    }
    return new If(e, s1, s2);
}
```

구현 (실습)

나 때라서면 파서까지 다 했음.

파서 구현: While 문

- 구문법

  <whileStatement> → while (<expr>) <stmt>

- 파서 구현

```
While whileStmt ( ) {
    match(Token.WHILE);      // while 토큰 매치
    match(Token.LPAREN);     // 왼쪽 괄호 매치
    Expr e = expr( );        // 수식 파싱
    match(Token.RPAREN);     // 오른쪽 괄호 매치
    Stmt s = stmt( );        // 본체 문장 파싱
    return new While(e, s);  // AST 구성 및 리턴
}
```

let-문 파싱

- 구문법

  <letStatement> → let <decls> in <stmts> end

- 파서 구현

```
Let letStatement () {
    match(Token.LET);          // let 토큰 매치
    Decls ds = decls( );       // 변수 선언 파싱
    match(Token.IN);           // in 토큰 매치
    Stmts ss = stmts( );  // 본체 문장들 파싱
    match(Token.END);   // end 토큰 매치
    match(Token.SEMICOLON);    // 세미콜론 매치
    return new Let(ds, null, ss);    // AST 구성 및 리턴
}
```

파서 구현: 수식

- 구문법

  <aexp> → <term> {+ <term> | - <term>}

- 파서 구현
  - 수식을 파싱하고 수식을 위한 AST를 구성하여 리턴한다.

```
Expr aexp() {
    Expr e = term( );              // 첫번째 항(term) 파싱
    while (token==Token.PLUS || token== Token.MINUS) {   // + 혹은 -
        Operator op = new Operator(match(token));      // 연산자 매치
        Expr t = term( );          // 다음 항(term) 파싱
        e = new Binary(op, e, t);   // 수식 AST 구성
    }
    return e;                      // 수식 AST 리턴
}
```

파서 구현: term 함수

- 구문법

  <term> → <factor> {* <factor> | / <factor>}

- 파서 구현
  - 항(term)을 파싱하고 항(term)을 위한 AST를 구성하여 리턴한다.

```
Expr term () {
    Expr t = factor( );          // 첫번째 인수(factor) 파싱
    while (token==Token.MULTIPLY || token==Token.DIVIDE) { // * 혹은 /
        Operator op = new Operator(match(token));        // 연산자 매치
        Expr f = factor( );      // 다음 인수(factor) 파싱
        t = new Binary(op, t, f);    // 항의 AST 구성
    }
    return t;                    // 항의 AST 리턴
}
```

파서 구현: factor 함수

<factor> → [-] (number | '('<aexp>')' | id)

```
Expr factor() {
    Operator op = null;
    if (token == Token.MINUS)
        Operator op = new Operator(match(token));  // 단항 – 연산자 매치
     Expr e = null;
     switch(token) {
     case ID:
        Identifier v = new Identifier(match(Token.ID));  // 식별자 매치
        e = v;  break;
     case NUMBER: case STRLITERAL:
        e = literal( );  break;        // 정수 혹은 스트링 리터럴 파싱
     case LPAREN:
        match(Token.LPAREN);           // 왼쪽 괄호 매치
        e = expr( );                   // 괄호 수식 파싱
        match(Token.RPAREN);           // 오른쪽 괄호 매치
        break;
     default:  error("Identifier | Literal");
     }
     if (op != null) return new Unary(op, p);  // 단항 연산 AST 구성 및 리턴
     else return e;
}
```