Provided Files:

- Parser.java - A shell file
- i1.txt - A sample input file • i2.txt - A sample input file • i3.txt - A sample input file • i4.txt - A sample input file
- i5.txt - A sample input file
- w1.txt - The required output for i1.txt • w2.txt - The required output for i2.txt • w3.txt - The required output for i3.txt • w4.txt - The required output for i4.txt
- w5.txt - The required output for i5.txt

Description: Implement a parser for language **AC** (the same target language for Lexical Analyzer) using a parsing technique called *recursive descent*.

Consider production rules for **AC** below. `<Prog>` is the start variable.

```
<Prog> → <Dcls> <Stmts>
<Dcls> → <Dcl> <Dcls>
       | λ
<Dcl> → FLOATDCL ID
      | INTDCL ID
<Stmts> → <Stmt> <Stmts>
        | λ
<Stmt> → ID ASSIGN <Val> <Expr>
       | PRINT ID
<Expr> → PLUS <Val> <Expr>
       | MINUS <Val> <Expr>
       | λ
<Val> → ID
      | INUM
      | FNUM
```

A parser determines if a stream of lexemes provided by a lexical analyzer conforms to the target language's grammar specification. In recursive descent parsing, each non-terminal (i.e., variable) in production rules has an associated parsing function that is responsible for determining if an input program contains a sequence of lexemes derivable from that nonterminal. Each parsing function examines the next input lexeme to predict which production rule should be applied.

For example, `<Stmt>` offers two productions:

```
<Stmt> → ID ASSIGN <Val> <Expr>
       | PRINT ID
```

If `ID` is the next token, then the parser must proceed with a rule that generates a lexeme of type `ID` as its first terminal. Similarly, if `PRINT` is the next token, the second rule should be applied. If the next token is neither

`ID` nor `PRINT`, then neither rule can be predicted. Given that the function for `<Stmt>` is called only when the nonterminal `<Stmt>` should be derived, the input program must have a syntax error.

Now, consider the production rule for `<Stmts>`:

```
<Stmts> → <Stmt> <Stmts>
        | λ
```

Since the first production rule begins with the nonterminal `<Stmt>`, you must check if `ID` or `PRINT` is the next token, which predicts any rule for `<Stmt>`.

You may assume that:

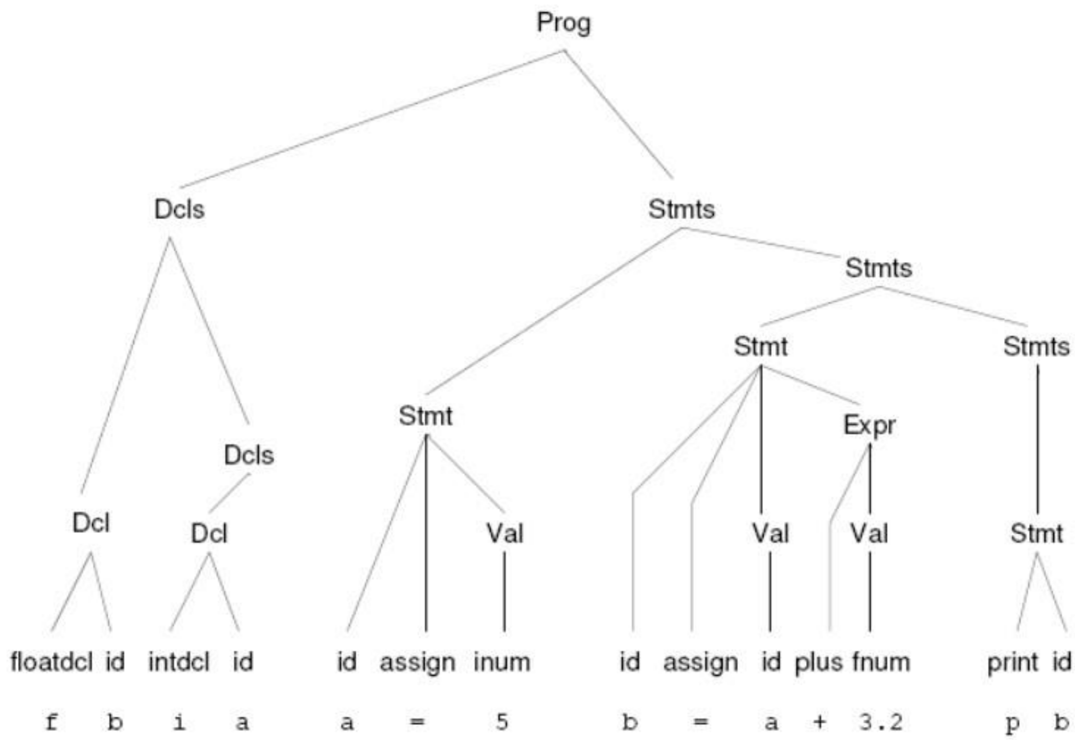
(1) the input program contains only legal lexemes,
(2) each lexeme is separated by a single blank, (3) there is no syntax error in the input program.

The output format follows the result of syntax analysis in the lecture slides – see **Appendix** on the next page.

Read the Appendix on the next page.

Appendix

Below is a parse tree for an input program: `f b i a a = 5 b = a + 3.2 p b`



The answer output is:

```
Next token is: 0, Next lexeme is f
Enter <Prog>
Enter <Dcls>
Enter <Dcl>
Next token is: 3, Next lexeme is b
Next token is: 1, Next lexeme is i
Exit <Dcl>
Enter <Dcls>
Enter <Dcl>
Next token is: 3, Next lexeme is a
Next token is: 3, Next lexeme is a
Exit <Dcl>
Exit <Dcls>
Exit <Dcls>
Enter <Stmts>
Enter <Stmt>
Next token is: 4, Next lexeme is =
Next token is: 7, Next lexeme is 5
Enter <Val>
Next token is: 3, Next lexeme is b
```

```
Enter <Stmts>
Enter <Stmt>
Next token is: 4, Next lexeme is =
Next token is: 3, Next lexeme is a
Enter <Val>
Next token is: 5, Next lexeme is +
Leave <Val>
Enter <Expr>
Next token is: 8, Next lexeme is 3.2
Enter <Val>
Next token is: 2, Next lexeme is p
Leave <Val>
Leave <Expr>
Leave <Stmt>
Enter <Stmts>
Enter <Stmt>
Next token is: 3, Next lexeme is b
Leave <Stmt>
Exit <Stmts>
Exit <Stmts>
```

Leave <Val> Leave <Stmt>	Exit <Stmts> Exit <Prog>