1 Concrete Syntax

```
Program    ::= { Definition }
Definition ::= { IDENT "=" Expression ";" }
Expression ::= Term { ("+"|"−") Term }
Term       ::= Factor { ("∗"|"/") Factor }
Factor     ::= IDENT | INTLIT | "(" Expression ")"
```


2 Parser

```
public Tree factor() throws IOException {
    switch(token) {
    case INTLIT: {
        token = nextToken();
        return null;
    }
    case IDENT: {
        token = nextToken();
        return null;
    }
```



```
    case LPAREN: {
        token = nextToken();
        expression();
        if (token == RPAREN) {
            token = nextToken();
        } else {
            Report.error(pos, "\')\' expected.");
        }
        return null;
    }
    default: {
        Report.error(pos, "\'(\', integer or identifier expected.");
        return null;
    }
    }
}
```

- Each time we have recognized a non-terminal, we need to call token = nextToken()

3 Abstract Syntax

```
Program    = PROGRAM { Definition }
Definition = DEF String Expression
Expression = IDENT String
           | INTLIT int
           | BINOP Expression Expression char
```

4 Tree Generation

```
public Tree factor() throws IOException {
    switch(token) {
    case INTLIT: {
        int i = Integer.parseInt(chars);
        int p = pos;
        token = nextToken();
        return new IntLiteral(p, i);
    }
    case IDENT: {
        String name = chars;
        int p = pos;
        token = nextToken();
        return new Ident(p, name);
    }
```



```
    case LPAREN: {
        token = nextToken();
        Tree expr = expression();
        if (token == RPAREN) {
            token = nextToken();
        } else {
            Report.error(pos, "\')\' expected.");
        }
        return expr;
    }
    default: {
        Report.error(pos, "\'(\', integer or identifier expected.");
        return new Bad(pos);
    }
    }
}
```


5 Tree Generation (2)

```
public Tree term() throws IOException {
    Tree t = factor();
    while (token == TIMES || token == DIV) {
        char op;
        if (token == TIMES) {
            op = '*';
        } else {
            op = '/';
        }
        int p = pos;
        token = nextToken();
        t = new BinOp(p, op, t, factor());
    }
    return t;
}
```


6 Print Visitor

```
protected void print(Tree tree) {
    tree.apply(this);
}
public void caseDef(Def tree) {
    System.out.print(tree.name + " = ");
    print(tree.expr);
    System.out.print(";");
}
public void caseIdent(Ident tree) {
    System.out.print(tree.name);
}
```

- We use print to print subtrees in the abstract syntax tree.
- We use System.out.print to print the leaves of the abstract syntax tree.

```
public void caseIntLiteral(IntLiteral tree) {
        System.out.print("" + tree.i);
}
public void caseBinOp(BinOp tree) {
        System.out.print("(");
        print(tree.left);
        System.out.print(" " + tree.op + " ");
        print(tree.right);
        System.out.print(")");
}
```



```
PrintStream p;

public void caseDef(Def tree) {
        p.print(tree.name + " = ");
        print(tree.expr);
        p.print(";");
}
public void caseIdent(Ident tree) {
        p.print(tree.name);
}
```

- using a PrintStream instead of System.out makes our print-visitor mor flexible.