Concrete Syntax	Abstract Syntax
<Program> ::= prog IDENTIFIER <Block> gorp EOF	<Program> ::= IDENTIFIER <Block>
<Block> ::= (<Declaration>; | <Command>;)*	<Block> ::= <DecOrCommand> *
<Declaration> ::= <Type> IDENTIFIER	<Declaration> ::= <Type> IDENTIFIER
<Type> ::= <SimpleType> | <CompoundType>	
<SimpleType> ::= int | boolean | string	<SimpleType> ::= int | boolean | string (use enum in Kind class)
<CompoundType> ::= map [<SimpleType>, <Type>]	<CompoundType> ::=<SimpleType> <Type>
<Command> ::= <LValue> = <Expression>	<AssignExprCommand> ::= <LValue><Expression>
| <LValue> = <PairList>	<AssignPairListCommand> ::= <LValue><PairList>
| print <Expression>	<PrintCommand> ::= <Expression>
| println <Expression>	<PrintlnCommand> ::= <Expression>
| do (<Expression>) <Block> od	<DoCommand> ::= <Expression> <Block>
| do <LValue> : [IDENTIFIER , IDENTIFIER] <Block> od	<DoEachCommand> ::= <LValue> Identifier Identifier <Block>
| if (<Expression>) <Block> fi	<IfCommand> ::= <Expression> <Block>
| if (<Expression>) <Block> else <Block> fi	<IfElseCommand> ::= <Expression><Block> <Block>
| ε	
<LValue> ::= IDENTIFIER	<SimpleLValue> ::= IDENTIFIER
<LValue> ::= IDENTIFIER [<Expression>]	<ExprLValue> ::= IDENTIFIER <Expression>
<Pair> ::= [<Expression> , <Expression>]	<Pair> ::= <Expression> <Expression>
<PairList> ::= { <Pair> (, <Pair>)* } | { }	<PairList> ::= <Pair> *
<Expression> ::= <Term> (<RelOp> <Term>)* <Term> ::= <Elem> (<WeakOp> <Elem>)* <Elem> ::= <Factor> (<StrongOp> <Factor)*	<BinaryOpExpression>::= <Expression> op <Expression> (use enum in Kind class for Op)
<Factor>::= <LValue>	<LValueExpression> := <LValue>
<Factor>::= INTEGER_LITERAL	<IntegerLiteralExpression> ::= INTEGER_LITERAL
<Factor>::= BOOLEAN_LITERAL	<BooleanLiteralExpression> ::= BOOLEAN_LITERAL
<Factor>::= STRING_LITERAL	<IntegerLiteralExpression> ::= STRING_LITERAL
<Factor>::= (<Expression>)	
<Factor>::= ! <Factor>	<UnaryOpExpression> ::= op <Expression> (use enum in Kind class for Op)
<Factor>::= -<Factor>	<UnaryOpExpression> ::= op <Expression> (use enum in Kind class for Op)

<RelOp> ::= OR | AND | EQUALS | NOT_EQUALS | LESS_THAN | GREATER_THAN | AT_MOST | AT_LEAST <WeakOp> ::= PLUS | MINUS <StrongOp> ::= TIMES | DIVIDE	(use enum in Kind class for Op)

Recall that symbols with multiple rules map to an abstract class, instances of the rule are subclasses. Example:

```
abstract class Type{}
class SimpleType extends Type{…}
class CompoundType extends Type{…}
```