Abstract Syntax	Type Rules
<Program> ::= IDENTIFIER <Block>	The IDENTIFIER is a program name and cannot be used as a variable name
<Block> ::= <DecOrCommand> *	Each <Block> delineates a scope.
<Declaration> ::= <Type> IDENTIFIER	Condition: IDENTIFIER ∉ current scope. Insert name and declaration in symbol table. Clarification: Each identifier may only be defined once in the current scope. It may be redefined in a nested scope. Our language uses the most-closely-nested scope rule.
<SimpleType> ::= int | boolean | string (use enum in Kind class)	<SimpleType>.type ::= int or boolean or string
<CompoundType> ::=<SimpleType> <Type>	<CompoundType>.type ::= (keyType, valType) where keyType = <SimpleType>.type valType = <Type>.type
<AssignExprCommand> ::= <LValue><Expression>	Condition: <LValue>.type == <Expression.type> Example: x = y x and y must have the same type.
<AssignPairListCommand> ::= <LValue><PairList>	Condition let<LValue>.type = (keyType,valType) and <PairList>.type = (pairKeyType,valKeyType) in keyType == pairKeyType && valType == pairValType Example: if m = {[a,b]}, then w must have m.keyType == type of a && m.valType == type of b
<PrintCommand> ::= <Expression>	
<PrintlnCommand> ::= <Expression>	
<DoCommand> ::= <Expression> <Block>	Condition: <Expression>.type == boolean
<DoEachCommand> ::= <LValue> $IDENTIFIER_0$ $IDENTIFIER_1$ <Block>	Condition: let<LValue>.type = (keyType,valType) in $Identifier_0$t.type == keyType && $Identifier_1$.type == valType

	Example: do x:[a,b] ….od x must be defined with a map type, say (t1,t2) Then a must be defined and have type t1, and b must be defined with type t2.
<IfCommand> ::= <Expression> <Block>	Condition: <Expression>.type == boolean
<IfElseCommand> ::= <Expression><Block> <Block>	Condition: <Expression>.type == boolean
<SimpleLValue> ::= IDENTIFIER	Condition: IDENTIFIER ∈ symbol table and defined in current scope. <SimpleLValue>.type ≔ IDENTIFIER.type where the type of the IDENTIFIER is obtained from the symbol table
<ExprLValue> ::= IDENTIFIER <Expression>	Condition: IDENTIFIER ∈ symbol table and defined in current scope. Condition: let IDENTIFIER.type = (keyType,valType) in keyType == <Expression>.type <ExprLValue>.type ≔ valType Example: m[e] m is declared to be map of type (keyType, valType) e must have the same type as keyType. The type of m[e] is valType
<Pair> ::= <Expression> <Expression>	
<PairList> ::= <Pair> *	All pairs in the <PairList> have the same type. Example {[a,b],[1,c]} type of a is int type of b is same as type of c
<BinaryOpExpression>::= <Expression$_0$> op <Expression$_1$> (use enum in Kind class for Op)	see below
<LValueExpression> := <LValue>	<LValueExpression>.type ≔ <LValue>.type
<IntegerLiteralExpression> ::= INTEGER_LITERAL	<IntegerLiteralExpression>.type ≔ int

<BooleanLiteralExpression> ::= BOOLEAN_LITERAL	<BooleanLiteralExpression>.type ≔ boolean
<StringLiteralExpression> ::= STRING_LITERAL	<StringLiteralExpression>.type ::= string
<Expression>	Type is inferred given rules for each type of expression
<UnaryOpExpression> ::= op <Expression> (use enum in Kind class for Op)	Condition: op == - or op == ! && if op = - then <Expression>.type = int && if op = ! then <Expression>.type = boolean

Rules for BinaryOpExpressions

1. $<Expression_0>$.type == $<Expression_1>$.type unless the op is a + and one of them is a string and the other an int or boolean.
2. + can be applied to all types except boolean. The type is the type of the result. If one of the arguments is a string, then the result is a string.
3. ==, !=, >, <, ≤, ≥ apply to any type and the result is boolean
4. * and - can be applied to integers and maps. The result is the same as the argument type
5. / can be applied to integers, the result is the same as the argument type.
6. & and | can be applied to boolean types, the result is a boolean.