COP5556 Assignment 4

Create an ASTVisitor class that will traverse the AST created in Assignment 3 and perform type checking. The concrete syntax remains for convenience, but the type rules apply to the abstract syntax. You will also need to create an appropriate symbol table, which maps names to their Declaration.

|  |  |  |
| --- | --- | --- |
| Concrete Syntax | Abstract Syntax | Type constraints |
| Program ::= ( Declaration SEMI | Statement SEMI )\* | Program ::= (Dec | Statements)\* |  |
| Declaration :: = VariableDeclaration | ImageDeclaration | Dec ::= DecVar | DecImage |  |
| VariableDeclaration ::= VarType IDENT ( ASSIGN Expression | ϵ ) | DecVar ::= Type IDENT (Expression | ϵ ) | Constraints:   * name (of IDENT) has not been previously declared. * Type == Expression.type (11/ 11:08am. Was =)   Effect: (name,dec) added to symbol table |
| VarType ::= KW\_int | KW\_string | Type ::= Int | String | Image | Boolean | Void |  |
| ImageDeclaration ::= KW\_image (LSQUARE Expression COMMA Expression RSQUARE | ϵ) IDENT (((LARROW | ASSIGN ) Expression) | ϵ ) | DecImage ∷= Type ( Expression0 Expression1 | ϵ) IDENT ( OP Expression2 | ϵ ) | Constraints:   * name has not been previously declared. * Expression0 is Int or Void * Expression1 is Int or Void * Expression0.type == Expression1.type (10/30 1:08pm) * If (OP = LARROW) (Expression2.type == String || Expression2.type == Image), else if (OP = ASSIGN) then Expression2.type == Image, else OP == NOP.   Effect: (name,dec) added to symbol table |
| Statement ::= AssignmentStatement | ImageOutStatement | ImageInStatement | LoopStatement | Statement ∷= StatementAssign | StatementOutFile | StatementOutScreen |StatementImageIn | StatementLoop | (Apply to all statements)  Constraints:   * name has been declared   Effect: dec is set to the Declaration  In the rules below, IDENT.type is the type specified for the corresponding name in its declaration. |
| ImageOutStatement ::= IDENT RARROW Expression | StatementOutFile ::= IDENT Expression | Constraints:   * IDENT.type == Image * Expression.type == String |
| ImageOutStatement ::=IDENT RARROW KW\_SCREEN ( LSQUARE Expression COMMA Expression RSQUARE | ϵ ) | StatementOutScreen ::= IDENT (Expression0 Expression1 | ϵ ) | Constraints:   * Expression0.type == Expression1.type * If (IDENT.type == Int || IDENT.type == String) then Expression0.type == Void * else if (IDENT.type == Image) then Expression0.type = Int or Void * else error. |
| ImageInStatement ::= IDENT LARROW Expression | StatementImageIn ::= IDENT Expression | Constraints:   * IDENT.type == Image * Expression.type == Image or String |
| AssignmentStatement ::= IDENT ASSIGN Expression | StatementAssign ::= IDENT Expression | Constraints:   * IDENT.type == Expression.type |
| LoopStatement ∷= IDENT ASSIGN STAR ConstXYSelector COLON (Expression | ϵ ) COLON Expression | StatementLoop ::= IDENT (Expression0 | ϵ ) Expression1 | Constraints:   * IDENT.type == Image * Expression0.type == Void or Boolean (11/1/20) * Expression1.type == Int |
|  | Expression | All Expressions have attribute type which is determined according to the rules below |
| Expression ::= OrExpression Q Expression COLON Expression | ExprConditional ::= Expression0 Expression1 Expression2 | Constraints:   * Expression0.type == Boolean * Expression1.type == Expression2.type   Result:   * ExprConditional.type = Expression1.type |
| Expression ::= OrExpression |  |  |
| OrExpression ::= AndExpression ( OR AndExpression)\* | BinaryExpr ::= Expression0 OP Expression1 | Constraints:   * If OP == AND or OR)   Expression0.type == Boolean  Expression1.type == Boolean  Result:   * BinaryExpr.type = Boolean |
| AndExpression ::= EqExpression ( AND EqExpression )\* |  |  |
| EqExpression ::= RelExpression ( (EQ | NEQ ) RelExpression )\* |  | Constraints:   * If OP == EQ or NEQ   Expression0.type == Expression1.type  Result:   * BinaryExpr.type = Boolean |
| RelExpression ::= AddExpression ( ( LT | GT | LE | GE ) AddExpression)\* |  | Constraints:   * If OP == LT, GT, LE, GE   Expression0.type == Expression1.type  Expression0.type == Int  Result:   * BinaryExpr.type = Boolean |
| AddExpression ::= MultExpression ( (PLUS | MINUS ) MultExpression )\* |  | Constraints:   * Expression0.type == Expression1.type * IF OP == PLUS, Expression0.type == Int or String * else if OP == MINUS Expression0.type = Int   Result:   * BinaryExpr.type == Expression0.type |
| MultExpression := UnaryExpression ( ( STAR | DIV | MOD ) UnaryExpression )\* |  | Constraints:   * Expression0.type == Expression1.type * Expression0.type == Int   Result:   * BinaryExpr.type = Int |
| UnaryExpression ::= (PLUS | MINUS) UnaryExpression | UnaryExpressionNotPlusMinus | ExprUnary ::= OP Expression | Constraints:   * If OP = PLUS or MINUS Expression.type == Int   Result   * ExprUnary.type = Int |
| UnaryExpressionNotPlusMinus ::= EXCL UnaryExpression | HashExpression |  | Constraints:   * if OP == EXCL Expression.type == Boolean   Result   * ExprUnary.type = Boolean |
| HashExpression ∷= Primary ( HASH Attribute)\* | ExprHash ::= Expression Attribute | Constraints:   * Expression.type == Int or Image * If Expression.type == Int, Attribute == “red”,”green”, or “blue” * else if Expressionltype == Image, Attribute == “width” or “height”   Result:   * ExprHash.type = Int |
| Primary ::= INTLIT | ExprIntLit | Result:   * ExprIntLit.type = Int |
| Primary ::= IDENT | ExprVar | Constraint:   * Name has been declared   Result:   * ExprVar.type = declared type |
| Primary ::= LPAREN Expression RPAREN |  |  |
| Primary ::= STRINGLIT | ExprStringLit | Result:   * ExprStringLit.type = String |
| Primary ::= KW\_X | KW\_Y | ExprVar | Result:   * ExprVar.type = Int   (Note: these variables are predefined, so create declarations and insert them into the symbol table during initialization of the Visitor. |
| Primary ::= CONSTANT | ExprConst | Result:   * ExprConst.type = Int |
| Primary ::= PixelConstructor |  |  |
| Primary ::= ArgExpression |  |  |
| Primary ::= (INTLIT | IDENT | LPAREN Expression RPAREN | STRINGLIT | KW\_X | KW\_Y | CONSTANT |PixelConstructor | ArgExpression ) (PixelSelector | ϵ ) | ExprPixelSelector ::= Expression ExpressionX  ExpressionY | Constraints:   * Expression.type == Image (11/1/20 at 6:36pm) * Expressionx.type == Expressiony.type == Int   Result:   * ExprPixelSelector.type = Int |
| PixelConstructor ∷= LPIXEL Expression COMMA Expression COMMA Expression RPIXEL | ExprPixelConstructor ::= Expressionr Expressiong Expressionb | Constraints:   * Expressionr.type == Expressiong.type == Expressionb.type == Int   Result:   * ExprPixelConstructor.type = Int |
| PixelSelector ∷= LSQUARE Expression COMMA Expression RSQUARE |  | (see above) |
| Attribute ∷= KW\_WIDTH | KW\_HEIGHT | KW\_RED | KW\_GREEN | KW\_BLUE |  |  |
| ArgExpression ∷= AT Primary | ExprArg ::= Expression | Contraints:   * Expression.type == Int   Result:   * ExprArg.type = expected type from context (which must be String or Int)   **See the hints** |
| ConstXYSelector ::= LSQUARE KW\_X COMMA KW\_Y RSQUARE |  |  |

* If type rules are violated, your TypeCheck visitor should throw a TypeException. The token where the error was manifested and an error message are parameters of the TypeException constructor. The contents of the message will not be graded, but you will appreciate it later if it is informative.
* If the given input is legal in the language, the visit method of the TypeCheckVisitor will return normally. Otherwise an exception may be thrown.
* ExprArg are different from every other expression type in that their attribute is inherited. These are expressions that get their values from command line arguments. For example int a = @0 will read the first command line argument (which is actually a String), convert it to an int and assign the value to a. Or string b = @1 would read the second command line argument and assign it to b. Thus the type of @0 and @1 depend on the context. The way to deal with this is to pass the expected type to the visit method of the ExprArg. For simplicity, we will only allow ExprArgs to appear in expressions where the type of the arguments is the same and the same as the type of the result. For example, a = @0 + @1; is OK (provided a is an Int or String), but a = (@0 == @1 ? 0 : 1); is not.