Grammar for Concrete Syntax (from previous assignments, included for completeness, you probably don't need it for this assignment)

Program::= Type IDENT (ParamList) Block

Block ::= <: (Declaration ; | Statement ;)* :>

ParamList ::= ε | NameDef (, NameDef) *

NameDef ::= Type IDENT | Type Dimension IDENT

Type ::= image | pixel | int | string | void | boolean

Declaration::= NameDef | NameDef = Expr

Expr::= ConditionalExpr | LogicalOrExpr

ConditionalExpr ::= ? Expr -> Expr , Expr

LogicalOrExpr ::= LogicalAndExpr ((| | ||) LogicalAndExpr)*

LogicalAndExpr ::= ComparisonExpr ((& | &&) ComparisonExpr)*

ComparisonExpr ::= PowExpr ((< | > | == | <= | >=) PowExpr)*

PowExpr ::= AdditiveExpr ** PowExpr | AdditiveExpr

AdditiveExpr ::= MultiplicativeExpr ((+ | -) MultiplicativeExpr)*

MultiplicativeExpr ::= UnaryExpr ((* | / | %) UnaryExpr)*

UnaryExpr ::= (! | - | width | height) UnaryExpr | PostfixExpr

PostfixExpr::= PrimaryExpr (PixelSelector | ε) (ChannelSelector | ε)

PrimaryExpr ::=STRING_LIT | NUM_LIT | IDENT | (Expr) | CONST | BOOLEAN_LIT |
ExpandedPixelExpr

ChannelSelector ::= : red | : green | : blue

PixelSelector ::= [Expr , Expr]

ExpandedPixelExpr ::= [Expr , Expr , Expr]

Dimension ::= [Expr , Expr]

LValue ::= IDENT (PixelSelector | ε) (ChannelSelector | ε)

Statement::=
LValue = Expr |
write Expr |
do GuardedBlock ([] GuardedBlock) * od |
if GuardedBlock ([] GuardedBlock) * fi |
^ Expr |
BlockStatement

GuardedBlock := Expr -> Block

BlockStatement ::= Block

Note: the rules with orange background were parsed in assignment 1.

Abstract Syntax

This is the grammar that describes the AST

Notation: A* means 0 or more instances of A
A^+ means 1 or more instances of A
$A^?$ means 0 or 1 instances of A

A* and A^+ are typically implemented using a List

$A^?$ typically means that the corresponding field in the AST node can be null.

Program::= Type IDENT NameDef* Block
Block ::= (Declaration | Statement)*
NameDef ::= Type $\text{Dimension}^?$ IDENT
Type ::= image | pixel | int | string | void | boolean
Declaration::= NameDef $\text{Expr}^?$
Expr::= ConditionalExpr | BinaryExpr | unaryOp Expr | PostFixExpr | PrimaryExpr | StringLitExpr | NumLitExpr | IdentExpr | | ConstExpr | BooleanLitExpr
ExpandedPixelExpr
ConditionalExpr ::= $\text{Expr}_{\text{GuardExpr}}$ $\text{Expr}_{\text{TrueExpr}}$ $\text{Expr}_{\text{FalseExpr}}$
BinaryExpr ::= $\text{Expr}_{\text{leftExpr}}$ op $\text{Expr}_{\text{rigthExpr}}$
UnaryExpr ::= op Expr
PostfixExpr::= Expr $\text{PixelSelector}^?$ $\text{ChannelSelector}^?$
ChannelSelector ::= red | green | blue
PixelSelector ::= $\text{Expr}_{\text{xExpr}}$ $\text{Expr}_{\text{yExpr}}$
ExpandedPixelExpr ::= Expr_{red} $\text{Expr}_{\text{green}}$ $\text{Expr}_{\text{blue}}$
Dimension ::= $\text{Expr}_{\text{width}}$ $\text{Expr}_{\text{height}}$
LValue ::= IDENT $\text{PixelSelector}^?$ $\text{ChannelSelector}^?$
Statement::=
AssignmentStatement |
WriteStatement |
DoStatement |
IfStatement |
ReturnStatement |
StatementBlock
AssignmentStatement ::= LValue Expr
WriteStatement ::= Expr
DoStatement ::= GuardedBlock^+
IfStatement ::= GuardedBlock^+
GuardedBlock := Expr Block
ReturnStatement ::= Expr
StatementBlock ::= Block

Context constraints

This is the grammar above annotated with attributes, rules and conditions for enforcing context constraints and which will later be used in code generation. Fields for attributes have been added to the classes representing the AST nodes.

The following symbols (which correspond to classes in the AST) have an attribute “type” which is implemented using the enum Type in the ast package.
Program, NameDef, Declaration, all of the Expr nodes.

The following have an attribute “nameDef” which has type NameDef: **LValue, IdentExpr**

The symbolTable implements block-structured lexical scoping that requires names to be declared before they are used, and names are visible when they are declared in the given scope or an enclosing scope. For this project, it is convenient to let the symbol table map names to the NameDef in their declaration. (One way to implement this semantics is using the LeBlanc-Cook symbol table described in class)

Note: Constraints apply to all nodes in the AST. This means that all children must be visited, even when this is not explicitly stated.

Abstract Syntax with context constraints

Program::= Type IDENT NameDef* Block
    Program.type ← Type
    symbolTable.enterScope()
    check children NameDef* and Block
    symbolTable.leave Scope()
    Note: there are no constraints involving IDENT—it is not entered into the symbol table
Block ::=
    symbolTable.enterScope()
    check children
    symbolTable.leaveScope()
NameDef ::= Type Dimension$^?$ IDENT
    Condition: if (Dimension != null) { type == IMAGE }
        else Type ∊ {INT, BOOLEAN, STRING, PIXEL, IMAGE}
    NameDef.type ← type
    symbolTable.insert(nameDef) is successful
Type ::= image | pixel | int | string | void | boolean
    Note: Implement with Enum Type in provided code

Declaration::= NameDef $\text{Expr}^?$

Condition: Expr == null
|| Expr.type == NameDef.type
|| (Expr.type == STRING && NameDef.type == IMAGE)
Declaration.type ← NameDef.type
Note: visit Expr before NameDef

Expr::= ConditionalExpr | BinaryExpr | unaryOp Expr | PostFixExpr | StringLitExpr | NumLitExpr | IdentExpr | | ConstExpr | BooleanLitExpr |ExpandedPixelExpr

ConditionalExpr ::= $\text{Expr}_{\text{guardExpr}}$
$\text{Expr}_{\text{trueExpr}}$
$\text{Expr}_{\text{falseExpr}}$
Condition: $\text{Expr}_{\text{guardExpr}}$.type == BOOLEAN
Condition: $\text{Expr}_{\text{trueExpr}}$.type == $\text{Expr}_{\text{falseExpr}}$.type
ConditionalExpr.type ← trueExpr.type

BinaryExpr ::= $\text{Expr}_{\text{leftExpr}}$ op $\text{Expr}_{\text{rigthExpr}}$ ‘

Condition inferBinaryType is defined
BinaryExpr.type ← inferBinaryType($\text{Expr}_{\text{leftExpr}}$.type, op, $\text{Expr}_{\text{rigthExpr}}$ .type)

inferBinaryType

$\text{Expr}_{\text{leftExpr}}$.type	**op**	**$\text{Expr}_{\text{rigthExpr}}$ .type**	**inferBinaryType**
PIXEL	BITAND, BITOR	PIXEL	PIXEL
BOOLEAN	AND, OR	BOOLEAN	BOOLEAN
INT	LT, GT, LE, GE	INT	BOOLEAN
any	EQ	$\text{Expr}_{\text{leftExpr}}$.type	BOOLEAN
INT	EXP	INT	INT
PIXEL	EXP	INT	PIXEL
Any	PLUS	$\text{Expr}_{\text{leftExpr}}$.type	$\text{Expr}_{\text{leftExpr}}$.type
INT,PIXEL,IMAGE	MINUS, TIMES, DIV, MOD	$\text{Expr}_{\text{leftExpr}}$.type	$\text{Expr}_{\text{leftExpr}}$.type
PIXEL,IMAGE	TIMES, DIV, MOD	INT	$\text{Expr}_{\text{leftExpr}}$.type

UnaryExpr ::= op Expr

Condition: inferUnaryExpr is defined

UnaryExpr.type ← inferUnaryExprType(Expr.type, op,)

inferUnaryExpr.type

Expr.type	**Op**	**inferUnaryExprType**
BOOLEAN	BANG	BOOLEAN
INT	MINUS	INT
IMAGE	RES_width, RES_height	INT

PostfixExpr::= Expr PixelSelector? ChannelSelector?

Condition: inferPostfixExprType is defined

PostfixExpr.type ← inferPostfixExprType(Epxr.type, PixelSelector, ChannelSelector)

inferPostfixExprType

Expr.type	PixelSelector	ChannelSelector	inferPostfixExprType
Any	Null	Null	Expr.type
IMAGE	Not null	Null	PIXEL
IMAGE	Not null	Not null	INT
IMAGE	Null	Not null	IMAGE
PIXEL	Null	Not null	INT

StringLitExpr

StringLitExpr.type ← STRING

NumLitExpr

NumLitExpr.type ← INT

IdentExpr

Condition: symbolTable.lookup(IdentExpr.name) defined

IdentExpr.nameDef ← symbolTable.lookup(IdentExpr.name)

IdentExpr.type ← IdentExpr.nameDef.type

ConstExpr

ConstExpr.type ← if (ConstExpr.name == ‘Z’) INT else PIXEL

BooleanLitExpr

BooleanLitExpr.type <- BOOLEAN

ChannelSelector ::= red | green | blue

PixelSelector ::= $Expr_{xExpr}$ $Expr_{yExpr}$ *(see discussion below)*

If the PixelSelector's parent is an LValue then

Condition: $Expr_{xExpr}$ is an IdentExp or NumLitExpr

Condition: $Expr_{yExpr}$ is an IdentExp or NumLitExpr

If $Expr_{xExpr}$ is an IdentExp and symbolTable.lookup($Expr_{xExpr}$.name == null)

Insert a SyntheticNameDef with name $Expr_{xExpr}$.name

and type INT into the symbol table

end if

If $Expr_{yExpr}$ is an IdentExp and symbolTable.lookup($Expr_{yExpr}$.name == null)

Insert a SyntheticNameDef with name $Expr_{yExpr}$.name

and type INT into the symbol table

end if

end if

Condition: $Expr_{xExpr}$.type == INT

Condition: $Expr_{yExpr}$.type == INT

ExpandedPixelExpr ::= $Expr_{red}$ $Expr_{green}$ $Expr_{blue}$

Condition: $Expr_{red}$.type == INT

Condition: $Expr_{green}$.type == INT

Condition: $Expr_{blue}$.type == INT

ExpandedPixelExpr.type ← PIXEL

Dimension ::= $Expr_{width}$ $Expr_{height}$

Condition: $Expr_{width}$.type == INT

Condition: $Expr_{height}$.type == INT

LValue ::= $IDENT_{nameToken}$ $PixelSelector^{?}$ $ChannelSelector^{?}$

LValue.nameDef ← symbolTable.lookup(name)
LValue.varType ← LValue.nameDef.type
Condition: if (PixelSelector != null) LValue.varType == IMAGE
Condition: if (ChannelSelector != null) LValue.varType ∈ { PIXEL, IMAGE}
Condition: inferLValueType is defined
LValue.type ← inferLValueType

LValue.varType	**PixelSelector**	**ChannelSelector**	**inferLValueType**
any	null	null	LValue.varType
IMAGE	Not null	null	PIXEL
IMAGE	Not null	Not null	INT
IMAGE	null	Not null	IMAGE
PIXEL	null	Not null	INT

NOTE: when visiting nonnull PixelSelector, indicate somehow that context is in an LValue

Statement::=
AssignmentStatement |
WriteStatement |
DoStatement |
IfStatement |
ReturnStatement |
StatementBlock

AssignmentStatement ::= $LValue_{lValue}$ $Expr_{e}$

symbolTable.enterScope()
visit children to check condition
Condition: AssignmentCompatible (LValue.type, Expr.type)
symbolTable.leaveScope()

The function AssignmentCompatible is defined by the following table. Combinations not listed are false

LValue.type	**Expr.type**	**AssignmentCompatible**
Any	LValue.type	true
PIXEL	INT	true
IMAGE	PIXEL	true
IMAGE	INT	true
IMAGE	STRING	true

WriteStatement ::= Expr_{expr}

DoStatement ::= GuardedBlock^{+}

IfStatement ::= GuardedBlock^{+}

GuardedBlock := Expr_{Guard} Block_{block}
Condition: Expr.type == BOOLEAN

ReturnStatement ::= Expr_{e}
Condition: Expr.type == Program.type (where Program is the enclosing program)

StatementBlock ::= Block_{block}

More on PixelSelector:

Our language allows one deviation from normal rules for declaring variables. We are allowed to write assignment statements of the form

Image0[x,y] = Expr(x,y)

Where x and y are NOT already declared. To handle this case, we will implicitly declare them by entering a new scope and adding (x, SyntheticNameDef(x,INT)) and (y, SyntheticNameDef(y,INT)) to the symboltable before visiting the expression on the right hand side. The extent of the new scope is just the assignment statement, so after visiting Expr, we leave the scope. (When we generate code, we will have an implicit loop over x and y for all the pixels in the image.)

If the pixel selector appears on the right hand side, we just visit the children as normal and don't need to do anything special except check that they have the expected type.

Implementing this require that AssignmentStatements are in their own scope, and also that PixelSelectors have a way to determine whether or not they are in the context of an LValue. (Hint: one way is to use the arg parameter of the visit function)