Determine Analyzable Loop

An analyzable loop has the following definition.

> A for loop that has an induction variable (call it `i`) with:
>
> 1. one or more initialization statements
> 2. a single test expression that uses `<`, `<=`, `>` or `>=` operations
> 3. a single increment expression in the form `i=i+c`, `i=i-c`, `++i`, `--i`, `i++` or `++i`, where `c` is a compile-time constant
> 4. the value of `i` is not changed in the loop

Therefore, a combination of techniques is applied to check whether a loop is analyzable and find its induction variable if true.

At a high level, every loop, regardless it is nested or not, in the source file is checked separately. Every loop is analyzed by `SageInterface::isCanonicalForLoop` first. Then the step of the for loop is checked for whether it is a compile-time constant. Lastly, the candidate induction variable is examined for any write-reference to ensure its constness in the loop.

The `SageInterface::isCanonicalForLoop` API can check whether a for loop is canonical. A canonical for loop has the following definition.

> A canonical form is defined as:
>
> 1. one initialization statement
> 2. a test expression (a single test expression that uses `<`, `<=`, `>` or `>=` operations)
> 3. an increment expression
> 4. loop index variable should be of an integer type

In other words, the check performed by this API covers item 1, 2 and 3 from the definition of analyzable loop, with two exceptions. The first exception is that it does not check whether or not the increment step `c` is a compile time constant, this can be addressed later with the `isSgIntVal` check. The second exception actually causes some headaches, that it only accepts one initialization statement, as opposed to the "or more" rule stated in the definition of analyzable loop. So in this case, if the init-statement has multiple init-expressions that are separated by the comma operator, the loop will not be accpeted as analyzable, which it should according to the definition of analyzable loop. However, if I were to abide with the proper definition, then I would need to reimplement the entire functionalities provided by the `SageInterface::isCanonicalForLoop` API, and could introduce other undesired behaviors. So I decided to violate the proper definition of analyzable loop and continue to use the `SageInterface::isCanonicalForLoop` API.

Next, the increment step `c`, which is acquired from the `SageInterface::isCanonicalForLoop` API as `SgExpression`, is checked for whether it is a compile-time constant. It is tested with the `isSgIntVal` API that checks whether the expression represents an integer literal. However, this may not be able to accept other compile time constant cases that involve constant propagation and constant folding, such as a constant variable defined to be `const int step = 2;` or in-place constant arithmetic expression `i += 1+1`. A possible solution is to perform a constant propagation and constant folding optimization pass ahead of the loop analysis, which is beyond the scope of this assignment.

Lastly, a list of all variables that have read and/or write references is retrieved from the
`SageInterface::collectReadWriteVariables` API. The candidate induction variable, which is also
acquired from the the `SageInterface::isCanonicalForLoop` API, is checked against this list to find
any write-references inside the loop.

Table of Data Dependence Testing Problem Pairs

Test	Type	Dependence Entry

testA.c	*WR*	`out[i][j] : out[i][j] : t`
		`in[i][j] : in[i - 1][j] : t`
		`in[i][j] : in[i + 1][j] : t`
		`in[i][j] : in[i][j - 1] : t`
		`in[i][j] : in[i][j + 1] : t`

testB.c	*WR*	`a[i][j] : a[i - 1][j] : t, i, j`
		`a[i][j] : a[i + 1][j] : t, i, j`
		`a[i][j] : a[i][j - 1] : t, i, j`
		`a[i][j] : a[i][j + 1] : t, i, j`

testC.c	*WR*	`b[i + 1][j + k + 1][k + 1] : b[i][j][k] : k, i, j`
testD.c	*WR*	`a[i + 1] : a[i] : i`
testE.c	*WR*	`b[i][index] : b[i - 1][index - 1] : i, j`
	WW	`eps[index] : eps[zoneset[i]] : i` `eps[index1] : eps[index2] : i`
testF.c	*WR*	`c[i][j] : c[i][j] : i, j, k`
testG.c	*WR*	`a[i] : a[i] : i` `a[i] : a[i] : i`