Context-free grammar for a subset of Quandary

<program> ::= <funcDefList>

<funcDefList> ::= <funcDef> <funcDefList>

| ε

<funcDef> ::= <varDecl> **(** <formalDeclList> **) {** <stmtList> **}**

<varDecl> ::= **int IDENT**

<formalDeclList> ::= <neFormalDeclList>

| ε

<neFormalDeclList> ::= <varDecl> **,** <neFormalDeclList>

| <varDecl>

<stmtList> ::= <stmt> <stmtList>

| <stmt>

<stmt> ::= <varDecl> **=** <expr> **;** // declare and initialize a variable

| **IDENT =** <expr> **;** // update an already-declared-and-initialized variable

| **if (** <condExpr> **)** <stmt>

| **if (** <condExpr> **)** <stmt> **else** <stmt>

| **while (** <condExpr> **)** <stmt>

| **{** <stmtList> **}** // nested block

| **return** <expr> **;**

<exprList> ::= <neExprList>

| ε

<neExprList> ::= <expr> **,** <neExprList>

| <expr>

<expr> ::= **INTCONST**

| **IDENT**

| **-** <expr>

| <binaryExpr>

| **(** <expr> **)**

| **IDENT (** <exprList> **)** // function call

<binaryExpr> ::= <expr> **+** <expr> // same for **-** and *****

<condExpr> ::= <expr> **<=** <expr> // same for **>=**, **==**, **!=**, **<**, and **>**

| <condExpr> **&&** <condExpr> // same for **||**

| **!** <condExpr>

| **(** <condExpr> **)**

Attribute grammar for the subset of Quandary

Attribute grammar goals (derived from the Quandary specification document):

1) Any occurrence of a variable in an expression must have a corresponding declaration of this variable in an earlier declaration (including declaration of formals and variable declarations in surrounding blocks). This allows code such as

int f(int x) { int y = 5; { int z = x+y; } }

2) A function must not declare a variable with the same name as another variable declared earlier in the same or an outer scope (including names of formal parameters). Scopes are demarcated by curly braces and by singleton statements in if-else and while statements.

3) Any occurrence of a function name in a function call expression must have a corresponding definition of this function somewhere in the program (thus, a call to a function could occur before this function is defined). The number/types of actual parameters at the call should match the number/types of formal parameters at the definition. The return type from the function definition should match the type of the expression using the return value at the call site (although this isn't really an issue with **int** being the only type).

4) Any function name in a function definition must be different from the names of already-defined functions and from the names of built-in functions. For this current simplified subset of the language, the only relevant built-in function is **int randomInt(int)**.

5) There should be an **int main(int)** function in the program.

6) The last statement in a function body should be a **return** statement.

Attribute grammar style:

We will use pure attribute grammars. Attribute evaluation rules cannot have side effects.

Operations for the tree of symbol tables:

createRootSymbolTable() : returns a symbol table with no parent; this root symbol table represents the global scope in which functions are defined; the call returns a pointer to the new table. The symbol table is initialized with all built-in functions; for example, randomInt is mapped to type INT(INT).

x.createChildSymbolTable(tbl) : returns a symbol table containing the same entries as tbl, with parent table pointed-to by x

x.getRootSymbolTable() : returns the highest (root) ancestor of x

x.clone(decl): returns a new table that has the same parent (if any) as table x and same entries as table x, plus the entry (if any) in decl

x.lookupId(y): starting with the table pointed-to by x, checks whether there is an id y. Returns the type of the id in the table. If not defined, check in the parent table; if not defined there, check in the parent of the parent, etc. -- except do NOT check the root symbol table (which contains only function names). If not found, return NOTFOUND

Attributes:

<funcDefList>.tbl, <funcDef>.tbl, <formalDeclList>.tbl, <neFormalDeclList>.tbl, <stmtList>.tbl, <stmt>.tbl, <expr>.tbl, <binaryExpr>.tbl, <exprList>.tbl, <neExprList>.tbl, <condExpr>.tbl -- inherited; symbol table (map of string → type); represents the declarations that precede this non-terminal

<funcDefList>.decls -- synthesized; symbol table (map of string → type); represents functions declarations of this <funcDefList>

<funcDef>.decl -- synthesized; a pair of (string, type); represents the function declaration

<formalDeclList>.decls, <neFormalDeclList>.decls -- synthesized; symbol table (map of string → type); represents formal declarations of this <formalDeclList>

<formalDecl>.decl -- synthesized; a pair of (string, type); represents a variable declaration

<stmt>.decl -- synthesized; a set of 0 or 1 pairs of (string, type); represents 0 or 1 variable declarations

<formalDeclList>.typeList, <neFormalDeclList>.typeList, <exprList>.typeList, <neExprList>.typeList -- synthesized; a list of variable types; the list of types in a formal declaration list or function call expression list

<varDecl>.id -- synthesized; a string; the name of the declared variable or function

<varDecl>.type -- synthesized; a variable or function type; the type of the variable or function declaration

<expr>.type, <binaryExpr>.type -- synthesized; a variable type (Q, int, or Ref); the static type of the expression

Attribute grammar:

<program> ::= <funcDefList>

<funcDefList>.tbl := <funcDefList>.decls

<funcDefList> ::= <funcDef> <funcDefList>$_2$

Cond: [*<funcDef>.decl's id is not in <funcDefList>$_2$.decls****]***

<funcDefList>.decls := <funcDefList>$_2$.decls.clone(<funcDef>.decl)

<funcDef>.tbl := <funcDefList>.tbl

<funcDefList>$_2$.tbl := <funcDefList>.tbl

| ε

<funcDefList>.decls := createRootSymbolTable()

Cond: [*<funcDefList>.tbl.lookupId(main) = INT(INT)****]***

<funcDef> ::= <varDecl> **(** <formalDeclList> **) {** <stmtList> **}**

<funcDef>.decl := (<varDecl>.id, funcType(<varDecl>.type,<formalDeclList>.typeList))

<formalDeclList>.tbl := <funcDef>.tbl.createChildSymbolTable(<formalDeclList>.decls)

<stmtList>.tbl := <formalDeclList>.tbl.createChildSymbolTable(empty symbol table)

Cond: *Last stmt in <stmtList> is a return statement (need to figure out how to check this for the homework)*

Note 1: the formals of a function have their own symbol table, created by <funcDef>.tbl.createChildSymbolTable()

Note 2: helper function funcType constructs a representation of a function type, e.g., given return type INT and list of parameter types INT,INT it produces INT(INT,INT)

<varDecl> ::= **int IDENT**

<varDecl>.id := IDENT.lexval

<varDecl>.type := INT

<formalDeclList> ::= <neFormalDeclList>

<formalDeclList>.typeList := <neFormalDeclList>.typeList

<formalDeclList>.decls := <neFormalDeclList>.decls

| ε

<formalDeclList>.typeList := emptyList()

<formalDeclList>.decls := empty symbol table

<neFormalDeclList> ::= <varDecl> **,** <neFormalDeclList>$_2$

Cond: [*<neFormalDeclList>$_2$.decls does not contain <varDecl>.id****]***

<neFormalDeclList>.decls := <neFormalDeclList>$_2$.decls union { (<varDecl>.id, <varDecl>.type) }

<neFormalDeclList>.typeList := prepend(<varDecl>.type, <neFormalDeclList>$_2$.typeList)

| <varDecl>

<neFormalDeclList>.typeList := prepend(<varDecl>.type, emptyList())

<neFormalDeclList>.decls := { (<varDecl>.id, <varDecl>.type) }

<stmtList> ::= <stmt> <stmtList>$_2$

<stmt>.tbl := <stmtList>.tbl

<stmtList>$_2$.tbl := <stmtList>.tbl.clone(<stmt>.decl)

| <stmt>

<stmt>.tbl := <stmtList>.tbl

<stmt> ::= <varDecl> **=** <expr> **;**

Cond: [*<varDecl>.type = <expr>.type****]***

<expr>.tbl := <stmt>.tbl

Cond: [*<stmt>.tbl.lookupId(<varDecl>.id) = NOTFOUND****]***

<stmt>.decl = { (<varDecl>.id, <varDecl>.type) }

| **IDENT =** <expr> **;**

Cond: [*<stmt>.tbl.lookupId(IDENT.lexval) = <expr>.type****]***

<expr>.tbl := <stmt>.tbl

<stmt>.decl = empty symbol table

| **if (** <condExpr> **)** <stmt>$_2$ // similarly for if-then-else and while

<condExpr>.tbl := <stmt>.tbl

<stmt>$_2$.tbl := <stmt>.tbl

<stmt>.decl = empty symbol table

| **{** <stmtList> **}**

<stmtList>.tbl := <stmt>.tbl.createChildSymbolTable(empty symbol table)

<stmt>.decl = empty symbol table

| **return** <expr> **;**

Cond: [*<expr>.type should match the return type of the function; details not shown****]***

<expr>.tbl := <stmt>.tbl

<stmt>.decl = empty symbol table

<expr> ::= **INTCONST**

<expr>.type := INT

| **IDENT**

Cond: [*<expr>.tbl.lookupId(IDENT.lexval) ≠ NOTFOUND****]***

<expr>.type := <expr>.tbl.lookupId(IDENT.lexval)

| **-** <expr>$_2$

Cond: [*<expr>$_2$.type = INT****]***

<expr>.type := INT

<expr>$_2$.tbl := <expr>.tbl

| <binaryExpr>

<expr>.type := <binaryExpr>.type

<binaryExpr>.tbl := <expr>.tbl

| **(** <expr>$_2$ **)**

<expr>.type := <expr>$_2$.type

<expr>$_2$.tbl := <expr>.tbl

| **IDENT (** <exprList> **)** // function call

Cond: [*<expr>.tbl.getRootSymbolTable().lookupId(IDENT.lexval) = some function type****]***

Cond: [*formalsTypes(<expr>.tbl.getRootSymbolTable().lookupId(IDENT.lexval)) = <exprList>.typeList****]***

<expr>.type := returnType(<expr>.tbl.getRootSymbolTable().lookupId(IDENT.lexval))

<exprList>.tbl := <expr>.tbl

<exprList> ::= <neExprList>

<exprList>.typeList := <neExprList>.typeList

<neExprList>.tbl := <exprList>.tbl

| ε

<exprList>.typeList := emptyList()

<neExprList> ::= <expr> **,** <neExprList>$_2$

<neExprList>.typeList := prepend(<expr>.type, <neExprList>$_2$.typeList)

<expr>.tbl := <neExprList>.tbl

<neExprList>$_2$.tbl := <neExprList>.tbl

| <expr>

<neExprList>.typeList := prepend(<expr>.type, emptyList())

<expr>.tbl := <neExprList>.tbl

<binaryExpr> ::= <expr>$_1$ **+** <expr>$_2$ // same for **-** and *****

Cond: [*<expr>$_1$.type = INT and <expr>$_2$.type = INT****]***

<binaryExpr>.type := INT

<expr>$_1$.tbl := <binaryExpr>.tbl

<expr>$_2$.tbl := <binaryExpr>.tbl

<condExpr> ::= <expr>$_1$ **<=** <expr>$_2$ // same for **>=**, **==**, **!=**, **<**, and **>**

Cond: [*<expr>$_1$.type = INT and <expr>$_2$.type = INT****]***

<expr>$_1$.tbl := <condExpr>.tbl

<expr>$_2$.tbl := <condExpr>.tbl

| <condExpr>$_2$ **&&** <condExpr>$_3$ // same for **||**

<condExpr>$_2$.tbl := <condExpr>.tbl

<condExpr>$_3$.tbl := <condExpr>.tbl

| **!** <condExpr>$_2$

<condExpr>$_2$.tbl := <condExpr>.tbl

| **(** <condExpr>$_2$ **)**

<condExpr>$_2$.tbl := <condExpr>.tbl