AGRAJAG

A GRAph JArGon

Language Reference Manual

by:

Dongyang Jiang (dj2322)
Zachary Salzbank (zis2102)
Erica Sponsler (es3094)
Nate Weiss (ndw2114)

1. Introduction

AGRAJAG is a C-like programming language designed to allow the easy creation and manipulation of graphs. The purpose of our language is to simplify the process of representing connections between data points. Using this simple representation, the manipulation and extraction of data is also simplified.

2. Lexical conventions

i. Comments

A comment begins when the character sequence `/*` is encountered. The comment ends when the character sequence `*/` is encountered. Comments cannot be nested.

ii. Identifiers

An identifier can be comprised of any combination of alphabetic characters (upper and lower case), integers, and the underscore '_' character. Identifiers cannot begin with an integer or an underscore, but can be any length. Identifiers are case sensitive, for instance `helloWorld` and `HelloWorld` would not be equivalent.

iii. Keywords

The following words are reserved as keywords and cannot be used as identifiers:

- **`int`**
- **`char`**
- **`boolean`**
- **`Node`**
- **`void`**
- **`null`**
- **`return`**
- **`break`**
- **`if`**
- **`else`**
- **`while`**
- **`true`**
- **`false`**

iv. Constants

The following types of constants are supported in AGRAJAG:

- **Integer constants**
  Integers are any sequence of digits not enclosed within single or double quotes. Digits in any other context (such as in an identifier name) are not considered constants.
- **Character constants**
  A character constant is a single character contained within single quotes. A single quote character is denoted by \' (backslash + single quote), and a backslash character is denoted by \\ (two backslashes).

3. Syntax Notation

This reference manual uses the following syntax notation:

- Terminals: Indicated by **`bold Courier New`** typeface.
- Non-terminals: Indicated by regular `Courier New` typeface.
- Any terminal or non-terminal followed by the subscript $_{opt}$ notation, is not required and can be omitted.

4. Nodes

Nodes will be our primary data type.

1. **Data Structure**
   - A node will consist of a value and a list of pointers to that node's successors.
   - A node's value can be the value of any expression including another node.
   - A node is inherently directed, as not all nodes that are connected point to each other. A node's successors are not necessarily 'aware' that they are a child to the first node. However, if the user wishes to have an undirected graph, they must simply ensure that for every node `N`, all successors of `N` include `N` as a successor.
2. **Operations**
   - The value of each node will be mutable and obtainable through operations on that node.
     - For a node **`x`**, the value will be obtained by **`x.value`**
   - The list of successors will also be mutable and obtainable through operations on that node.
   - Each node will have internal functions to access the number of successors.
     - For a node **`x`**, the number of successors will be obtained by **`x.numSuccessors`**
   - For a node **`x`**, the n^{th} successor of this node will be obtained by `x[n]` where **n** is any integer.
3. **Intended Use**
   - Mainly to build trees and graphs, but other data structures can be created by using Nodes. An example is a string, which could be represented by a collection of nodes with character values.

5. Expressions

Expressions return values. This section lists the available expressions. Precedence of expressions is the same as the order of sections listed below. All expressions within a section have the same precedence.

1. **Primary Expressions**

   Primary expressions are the first to be evaluated. Leftmost expressions are evaluated first.

   a. `identifier`

   An identifier is any previously declared variable, as long as it has been declared according to the rules specified in the Declarations section below.

   b. `constant`

   Any boolean (**`true`** or **`false`**), integer, or character. The type returned for each type of constant is as follows:

- boolean: **`boolean`**
- integer: **`int`**
- character: **`char`**

c. **`(`** `expression` **`)`**

Parentheses are used to alter precedence. Expressions within parentheses will be evaluated as primary expressions, even if the expressions contained within the parentheses are a lower precedence than the surrounding operations. The returned type and value evaluates to the same as `expression`.

d. `function-name` **`(`** `expression-list`$_{opt}$ **`)`**

Evaluates the expressions described in `function-name`, optionally passing values via `expression-list`. `expression-list` can be a single expression, or a comma-separated list of expressions. The declaration of `function-name` must return a value (cannot be a **`void`** function declaration). Calling **`void`** functions is covered under `statements`.

2. **Unary Operators**

Unary operators modify the result of a single expression. Rightmost unary operators are evaluated first.

a. **`–`** `expression`

The result is the negated value of `expression`, with the same type. `expression` must be **`int`**.

b. **`!`** `expression`

The result is the logical opposite of `expression`. `expression` must be of type **`boolean`** and the return type is **`boolean`**.

3. **Multiplicative Operators**

Leftmost multiplicative operators are evaluated first.

a. `expression` **`*`** `expression`

Multiplies the first `expression` by the second `expression`. The type of both expressions must be the same. The allowed type for `expression` is **`int`**.

b. `expression` **`/`** `expression`

Divides the first `expression` by the second `expression`. The type restrictions for multiplication apply to division as well. Since division is always integer division, the result will be the highest integer value less than or equal to the quotient.

4. **Additive Operators**

Leftmost additive operators are evaluated first.

a. `expression` **`+`** `expression`

Adds the first `expression` to the second `expression`. Allowed types for `expression` are **`int`** or **`char`**, but both `expressions` must be of the same type.

b. `expression` **`-`** `expression`

Subtracts the second `expression` from the first `expression`. The type restrictions for addition apply to subtraction as well.

5. **Relational Operators**

Relational operators return a **boolean** type. Allowed types for `expression` are **int** or **char**, but both `expressions` must be of the same type. The following relational operators are available:

- `expression` **<** `expression` (less than)
- `expression` **>** `expression` (greater than)
- `expression` **<=** `expression` (less than or equal to)
- `expression` **>=** `expression` (greater than or equal to)

6. **Equality Operators**

   Equality operators return a **boolean** type. The type restrictions for relational operators apply to equality operators as well. The following equality operators are available:

   - `expression` **==** `expression` (equal to)
   - `expression` **!=** `expression` (not equal to)

7. `expression` **&&** `expression`

   The **boolean and** operator (**&&**) returns **true** if both expressions are true. Otherwise, it returns **false**. Both expressions must be of **boolean** type.

8. `expression` **||** `expression`

   The **boolean or** operator (**||**) returns **true** if either expression is true. Otherwise, it returns **false**. Both expressions must be of **boolean** type.

9. **Assignment Operator**

   An assignment has the following form:

   ```
   identifier = expression or
   node-identifier.value = expression
   node-identifier[expression] = expression
   ```

   - Note: the `expression` within the brackets must evaluate to an `int`

   The rightmost assignment will occur first. Operands must have the same type. After the leftmost assignment occurs, the value of `expression` is returned.

6. Declarations

Declarations are used in AGRAJAG to specify variables and functions. When declaring a variable the type and name must be specified. The types available are:

- **int**
- **char**
- **boolean**
- **Node<Type>**

  note: when using a `Node`, the contained type specified can be any of the above types, including `Node`

Each variable must have its own type specified, and must be a separate statement from other declarations. After a variable declaration has been made wherever the variable appears in the program, the value associated with the variable will be used. A variable declaration has three parts: type, variable name, statement end. For instance, declaring a variable called number of type int would look like this:

```
int number;
```

Variables may also be instantiated at the same time they are declared. The same rules for assignment apply as in the section above. The declaration/instantiation would have the form:

```
int number = expression;
```

Functions can be declared to return any of the types listed above, or the `void` type, and can take as parameters any values and/or variables of the types listed above. Function declarations can be made anywhere in the program, before or after the function(s) that call them, but cannot be nested within other function definitions. Declarations of functions have four parts: return type, function name, parameter list, function body. Return type specifies the type of value the function returns. Function names can be any valid identifier that is not a reserved keyword. Parameter lists can have an arbitrary number of parameters with any mixture of the above types. A function body consists of a set of statements enclosed in braces ('`{`', '`}`'). These statements can be any valid AGRAJAG program, but must return a value of the same type as the specified return type. The form of a function declaration is:

`type function_name(`*`parameter-list`*`) {`*`sequence of statements return statement`*`}`

where *`parameter-list`* is a comma separated list of zero or more variables and their associated types. For instance:

```
type parameter1, type parameter2, type parameter3
```

The `root` function is the entry point to the program. It will be the first function executed, and has the return type `void`.

7. Statements

There are several allowable statements in AGRAJAG. A sequence of statements will be executed in the order that they appear in the program. The statements that are recognized by AGRAJAG are:

- **Expression statement:** A single expression followed by the end-statement character, '`;`'.
- **Conditional statement:** An expression that evaluates a boolean expression, and will execute the appropriate statements based on the result. It has the form:

  `if (`*`boolean expression`*`) then {`*`sequence of statements`*`} else {`*`sequence of statements`*$_{\text{opt}}$`}`

  There is no form of conditional without an else section. If the programmer wishes there to be no action in the else case they may write a statement that has the form:

  `if (`*`boolean expression`*`) then {`*`sequence of statements`*`} else {}`

- **While statement:** An expression that evaluates a sequence of expressions based on the value of a boolean expression. The form of a while statement is:

  `while (`*`boolean expression`*`) {`*`sequence of statements`*`}`

  The sequence of statements will be executed until the boolean expression evaluates to false. The boolean expression will be evaluated before each execution of the loop statements.

- **Return statement:** A statement that specifies what value a function should return. A return statement has the form:

  `return (`*`expression`*$_{\text{opt}}$`);`

  The expression inside the return statement will be evaluated, and the value will then be returned to the calling function. The type of the evaluated expression must match the return type of the function in which the return statement appears. If the expression is omitted, then no value will be returned. In this case, the type of the function should be `void`.

- **Calling a function** with a return type of **`void`** is a statement as well, because all expressions must evaluate to a value, but statements do not have this restriction. A function call to a void function has the form:

  `function-name(parameter-list`$_{\text{opt}}$`);`

8. Scope Rules

A lexical block begins with a '`{`' and ends with a '`}`'. The scope of any variable in a program will be the lexical block that it is defined in, after the point at which it is defined. If blocks are nested, the scope of the variables within the outer block extend into the inner block. To avoid ambiguity, no overlapping names can be used for identifiers; within any scope, there will only be one identifier with a certain name.

AGRAJAG allows for global variables, which are variables defined outside the scope of any one function, and will be in the scope of all functions. Nested functions are not allowed. Therefore, all functions have global scope. To avoid ambiguity, function identifiers must be unique.

9. Compilation and Output

Compilation on the file '`program.ag`' is performed by running the command:

```
./agrajag < program.ag
```

The program output will be displayed on the standard output of the terminal that the compiler was run on. The built in `print` function is used to output data. Any argument passed to the `print` function will be output.

10. Examples

```
void root(){
      Node<int> treeRoot = Node<5>;
      treeRoot[0] = Node<3>;
      treeRoot[1] = Node<7>;
      treeRoot[0][0] = Node<2>;
      treeRoot[0][1] = Node<4>;
      treeRoot[1][0] = Node<6>;
      treeRoot[1][1] = Node<8>;

      Node<int> result = binSearch(treeRoot, 4);
      if(result == null){
            print(false);
      } else {
            print(true);
      }

      result = bfs(treeRoot, 7);
      if(result == null){
            print(false);
      } else {
            print(true);
      }
      return ();
}

Node<int> binSearch ( Node<int> sNode, int searchFor ) {
      while (sNode != null) {
            if (searchFor < sNode.value) {
                  sNode = sNode[0];
            }
            else {
                  if(searchFor > sNode.value) {
                        sNode = sNode[1];
                  }
                  else {
                        return sNode;
                  }
            }
      }
      return null;
}

Node<int> bfs(Node<int> n, int target)
```

```
	{
		Node<Node<int>> toVisit = Node<n>;
		Node<Node<int>> endToVisit = toVisit;
		return bfsHelper(n, target, toVisit, endToVisit);
	}

	Node<int> bfsHelper(Node<int> n, int target, Node<Node<int>>
toVisit, Node<Node<int>> endToVisit)
	{
		if (n.Value == target) {
		return n;
		}
		else
		{}

		while (n[i] != null)
		{
			if(existsInGraph(n[i], toVisit)
			{
				Node<int> next = Node<n[i]>;
				endToVisit[0] = next;
				endToVisit = endToVisit[0];
			}
			else
			{}
			i = i + 1;
		}
		return bfsHelper(toVisit, target, toVisit[0], endToVisit);
	}

	boolean existsInGraph(Node<int> n, Node<Node<int>> graph)
	{
		Node<Node<int>> temp = graph[0];
		while (temp != null)
		{
			if (temp == n)
			{
				return true;
			}
			temp = temp[0];
		}
		return false;
	}
```

Appendix I - Syntax Summary

1. Expressions
    primary
    - expression
    ! expression
    expression binop expression
    lvalue assignop expression
2. Primary
    constant
    (expression)
    primary (expression-list$_{opt}$)
    lvalue
3. lvalue
    identifier
    node [primary]
    node.Value
4. binop
    + -
    / *
    < > <= >=
    != ==
    &&
    ||
5. assignop
    =
6. statement
    expression ;
    while (expression) { statement-list }
    if (expression) { statement-list } else { statement-list$_{opt}$ }
    return (expression);
7. statement-list
    statement
    statement statement-list
8. expression-list
    expression
    expression,expression-list