CA4003 - Compiler Construction
Assignment 2 Submission

Laura Araviciute 14324076

Programme:	Computer Applications
Module Code:	CA4003
Assignment Title:	Semantic Analysis and Intermediate Representation
Submission Date:	11/12/2017
Module Coordinator:	David Sinclair

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.
I/We have read and understood the referencing guidelines found at

http://www.dcu.ie/info/regulations/plagiarism.shtml ,
https://www4.dcu.ie/students/az/plagiarism and/or recommended in the assignment guidelines.

Name(s): Laura Araviciute **Date**: 11/12/2017

Semantic Analysis

Abstract Syntax Tree

I used a combination of lecture notes and the following external resources to help me with the AST
https://web.cs.wpi.edu/~cs544/PLTprojectast.html
https://www.youtube.com/watch?v=9V4GzEomc5w
https://www.cs.nmsu.edu/~rth/cs/cs471/InterpretersJavaCC.html
https://stackoverflow.com/questions/13902239/how-to-implement-jjtree-on-grammar
http://cs.lmu.edu/~ray/notes/javacc/
http://www.softwaresecretweapons.com/jjtree.html

In order to implement the AST, I took a systematic approach. I chose an example program and walked the program myself by seeing what nodes it may create. Firstly I traced through the Prog node and traced every possible step through the non terminals down to the very last node that produced a terminal symbol. In order to get the value of some of the nodes such as declarations etc I implemented some terminals as non terminals such as Identifier() and Type().

```
String Type() #Type : {Token t;}
{
    t = <INTEGER> {jjtThis.value = t.image; return t.image; }
|   t = <BOOLEAN>  {jjtThis.value = t.image; return t.image; }
|   t = <VOID>  {jjtThis.value = t.image; return t.image; }
}
```

This made sure that i could keep that of ID's and Types later in semantic analysis.
I created nodes for everything I thought would be useful later in semantics trying to give myself as much information as possible. I also changed the printing of the nodes to include the value if it had one associated with it so I could see it better.
As I played around with semantic analysis, I ended up removing a few nodes that gave me redundant information.

Symbol Table

In order to implement a symbol table to my language I had to analyse the language and decide on what sort of information would be useful to see in a symbol table.
Here are some of the external references I used:
http://www.cs.cornell.edu/courses/cs412/2008sp/lectures/lec12.pdf
https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm

All of the information found had suggested that using Hashtables was the most efficient route to take. I decided that I wanted to keep track of the following information:

- Scope : ie. global/main/function
- Type: ie. the type meaning integer/void/boolean (declarations and statements)
- Description: ie parameter of a function/function/variable/const

The lecture notes and the external sources have suggested external chaining which is why I had to implement 3 separate hashTables inside my SymbolTable object.

The first is the main symbol table which will be identified with a key of “global” since every variable and function start off with “global” declaration. Scope only changes as you move down the program.
Upon initialisation, a new hashtable with the starting scope of global and an empty list of scope ID’s.
A type table is used to store all of the types for different scopes. The keys to a type table is the combination of scope+id. The same is done with the description table.

In simple terms, SymbolTable object has a symbol table of keys which are the scopes (ie. function names, global, main). Each key in the symbol table has an associated LinkedList of values which contain the ID of the items stored in each scope. If a list associated with a particular scope has already associated non empty linked list, we add the new list to the top of the list so that is the most recent invocation of the ID. The lecture notes iterate that when a binding is made it has to be added to the top. This will later be useful in semantic checking. For each ID in each scope, the associated types and descriptors can be found using indexing with key of id+scope.

```
public void put(String id, String type, String information, String scope) {
    // see if current scope already exists
    LinkedList<String> list = symbolTable.get(scope);
    if(list == null) {
        list = new LinkedList<>();
        // new scope with id
        list.add(id);
        // add to list of scopes available in global scope
        symbolTable.put(scope, list);
    }
    else {
        // add as first element as we want most current value for later semantic
        analysis
        list.addFirst(id);
    }
    // unique has for each is id+type / id+information
    typeTable.put(id+scope, type);
    descriptionTable.put(id+scope, information);
}
```

In order to traverse the symbol table, we get all the keys of the parent symbol table (ie. all scopes). For each scope we grab the associated LinkedList of ID's in the scope. For each ID within the scope we print out it's ID, type and description.

In order to implement this symbol table in the jjTree, I initialised a SymbolTable as a global object alongside a String of default scope which is "global".

For each declaration (ie. var dec/ const declaration). I added the appropriate ID's, types and gave a string description of the declaration. I used the default scope, which means that whatever the scope is set to at this point in the program, that's the scope for the corresponding variable declaration.

```
void VariableDeclaration() #VarDec : {Token t; String id; String type;}
{
    t = <VARIABLE> id = Identifier() <COLON> type = Type()
    {
        jjtThis.value = t.image;
        symbolTable.put(id, type, "var", scope);
    }
}
```

For functions, the process is the same except for the fact that after we add to the symbolTable, we change the scope to the name of the function as the scope changes as soon as we enter a function. The function inside the symbol table will contain the scope of whatever it was previous before the enter to the function. We also check to see if we are not already in the global scope, to add the function to the global scope also as every function should also be in the global scope. This was a bug found later in the code. I am presuming however that every function declared is global in scope and that no functions are allowed to be declared within other functions.

```
void Function() #Function : {Token t; String type; String id;}
{
    // add the function into global scope also
    (type = Type() id = Identifier() {symbolTable.put(id, type, "function", scope);

    if(!scope.equals("global")) {
        symbolTable.put(id, type, "function", "global");
        }
     scope = id;} <LBRACKET> ParameterList() <RBRACKET> <IS>
    DeclarationList()
    <BEGIN>
    StatementBlock()
    <RETURN> <LBRACKET> (Expression() | {}) <RBRACKET> <SEMICOLON> #Return
    <END>)
    // we have entered new scope here
    // scope of that function is still whatever scope it was previously
}
```

Alongside the default functions of adding and traversing the symbol table, As I started semantic analysis I added a few other helper methods.

```
public boolean noDuplicates(String id, String scope) {
    LinkedList<String> list = symbolTable.get(scope);
    LinkedList<String> global_list = symbolTable.get("global");
    if(scope.equals("global")) {
        return global_list.indexOf(id) == global_list.lastIndexOf(id);
    }
    return ((list.indexOf(id) == list.lastIndexOf(id)) && (global_list.indexOf(id
    ) == -1));

}
```

The above function checks the symbol table for duplicates. If the scope in question is not global, it'll check that scope as well as the global scope. If the scope in question is just global it will only check the global scope.

I also implemented some getter methods as needed to get the description, type and scope tables for semantic analysis.

```
public String getType(String id, String scope) {
    String type = typeTable.get(id+scope);
    if(type != null)
        return type;
    else {
        type = typeTable.get(id+"global");
        if(type != null) {
            return type;
        }
    }
    return null;
}
```

When getting type/description etc I also checked the global scope if it's not available in the scope it is searched for.

Semantic Analysis

In order to implement semantic analysis, I implemented a SemanticAnalyser which implements the CCALParserVisitor. This will visit each note recursively where I will perform a set of semantic checks.
I set a global variable of scope within the semantic analyser to keep track of the scope as I'm moving through the program. Scope starts with "global" and if it moves into a function or main it changes accordingly. The semantic analyser uses the symbol table for it's checks.

Is no identifier declared more than once in the same scope?

In order to keep track of duplicate declarations of a variable, I have implemented a duplicate check every time a VAR/CONST node is entered.

```
private static void duplicateCheck(String id, String scope) {
  if(!ST.noDuplicates(id, scope)) {
      HashSet<String> dups = duplicates.get(scope);
      if(dups == null) {
          LinkedHashSet<String> set = new LinkedHashSet<>();
          set.add(id);
          duplicates.put(scope, set);
      }
      else {
          dups.add(id);
      }
  }
}
```

Firstly, the method uses the Symbol Table method of checking if there exists duplicates within the scope we are in at the given point in time. It then also checks the global scope for multiple declarations also. If it does contain duplicates, the duplicates are added in a LinkedHashSet with the id and the scope it was associated with. The LinkedHashSet will avoid storing duplicate values. The overall duplicates data structure is a HashTable with a string (scope) and the associated set of duplicates in the form of a LinkedHashSet. At the end of the program traversal of semantic checks, I perform a final checkForDuplicates() which traverses the duplicates hashTable and prints out all the duplicated variables or tells us that the program did not contain any duplicates.

```
public Object visit(VarDec node, Object data) {
  String id = (String)node.jjtGetChild(0).jjtAccept(this, data);
  String type = (String) node.jjtGetChild(1).jjtAccept(this, data);
  duplicateCheck(id, scope);
  return data;

}
private static void checkForDuplicates() {
    if(duplicates == null || duplicates.isEmpty()) {
        System.out.println("Program contains no duplicates");
    }
    else {
      Enumeration e = duplicates.keys();
      while(e.hasMoreElements()) {
          String scope = (String) e.nextElement();
          LinkedHashSet<String> dups = duplicates.get(scope);
          Iterator it = dups.iterator();
          System.out.print("Multiple declarations of [");
          while(it.hasNext()) {
             System.out.print(it.next());
          }

          System.out.println("] in " + scope);
    }
  }
}
```

Is every identifier declared within scope before its is used?

In order to check if a variable has been declared before use, I have implemented a isDeclared() function. The function checks the current scope and the global scope to see if they contain the declaration. The function is used within the Statement node entrances and error message is printed respectively.

```
private static boolean isDeclared(String id, String scope) {
    LinkedList<String> list = ST.getScopeTable(scope);
    LinkedList<String> global_list = ST.getScopeTable("global");
    if(list != null) {
        if(!global_list.contains(id) && !list.contains(id)) {
            return false;
        }
    }
    return true;
}
```

Are the arguments of an arithmetic operator the integer variables or integer constants?
Are the arguments of a boolean operator boolean variables or boolean constants?

I checked if a variable is being declared to see if it is a constant, if it is that is an error as constants cannot be redeclared. An error message is shown in that case. This will prevent any operations with the LHS of a constant.

```
if(isDeclared(id, scope)) {
    String type = ST.getType(id, scope);
    String description = ST.getDescription(id, scope);
    if(description.equals("const")) {
        System.out.println("ERROR: " + id + " is a constant and cannot be
        redeclared");
    }
```

Is the left-hand side of an assignment a variable of the correct type?

In order to check the correct assignment, I split the Statement node into two parts, the left hand side and right hand side. The “id” represents the left hand side of the statement. Now I traverse the other Statement node children to see if their right hand side matches.
For every “id”, i check the symbol Table to see what type it should be. If the type is “integer”, I check for the appropriate “Num” node, if it’s boolean, I check for “BoolOp” nodes. If the LHS and RHS do not match, an error message is written.
If the RHS is a function, I get the function name and check that the function is declared. I then check that the return type of the function is of the correct type also.

I do this for integers and booleans as they are the only data structures in the language.

```
String rhs = node.jjtGetChild(1).toString();
if(type.equals("integer")) {
    if(rhs.equals("Num"))
    {
        node.jjtGetChild(1).jjtAccept(this, data);
    }
    else if(rhs.equals("BoolOp")) {
        System.out.println("Expected type integer instead got boolean");
    }
    else if(rhs.equals("FuncReturn")) {
        String func_name = (String) node.jjtGetChild(1).jjtAccept(this, data);
        // check if function is declared in global scope
        if(!isDeclared(func_name, "global")) {
            System.out.println(func_name + " is not declared");
        }
        else {
            // get return type of function
            String func_return = ST.getType(func_name, "global");
            if(!func_return.equals("integer")) {
                System.out.println("Expected return type of integer instead
                got " + func_return);
            }
        }
```

```
else if(type.equals("boolean")) {
  if(rhs.equals("BoolOp"))
    {
        node.jjtGetChild(1).jjtAccept(this, data);
    }
    else if(rhs.equals("Num")) {
        System.out.println("Expected type boolean instead got integer");
    }
    else if(rhs.equals("FuncReturn")) {
        String func_name = (String) node.jjtGetChild(1).jjtAccept(this, data);
        // check if function is declared in global scope
        if(!isDeclared(func_name, "global")) {
            System.out.println(func_name + " is not declared");
        }
        else {
            // get return type of function
            String func_return = ST.getType(func_name, "global");
            if(!func_return.equals("boolean")) {
                System.out.println("Expected return type of boolean instead
                got " + func_return);
            }
        }
    }
}
```

Does every function call have the correct number of arguments?
Is there a function for every invoked identifier?

After having identified that a statement performs a function call that returns some value, I traverse through the FuncReturn's child ArgList which has the number of parameters passed

into the function. For the function, I search the symbol table to find out how many parameters the given function should have and compare against the number of children the ArgList node has. If the expected number of args and actual number of args do not match, an error message is written.

After that has been checked, I check to see if every parameter type matches the given parameter for the actual function. To do this, I have implemented a helper function in the symbol table data structure that will return the expected type of a parameter given it's index.

```
public String getParamType(int index, String scope) {
    int count = 0;
    LinkedList<String> idList = symbolTable.get(scope);
        for(String id : idList) {
            String type = typeTable.get(id+scope);
            String description = descriptionTable.get(id+scope);
            if(description.equals("param")) {
                count++;
                if(count == index) {
                    return type;
                }
            }
        }
        return null;
}
```

This given the index will return the type of parameter at a given index.

```
            int num_args = ST.getParams(func_name);
            // Statement -> FuncReturn -> ArgList -> children of arglist
            int actual_args = node.jjtGetChild(1).jjtGetChild(0).
            jjtGetNumChildren();
            // check that the correct number of args is used
            if(num_args != actual_args)
                System.out.println("ERROR: Expected " + num_args + " parameters
                instead got " + actual_args);
            else if(num_args == actual_args) {
                // check that the arguments are of the correct type
                Node arg_list = node.jjtGetChild(1).jjtGetChild(0);
                for(int i = 0; i < arg_list.jjtGetNumChildren(); i++) {
                    String arg  = (String)arg_list.jjtGetChild(i).jjtAccept(this,
                     data);
                    // check if argument in arglist is actually declared
                    if(isDeclared(arg, scope)) {
                        String arg_type = ST.getType(arg, scope);
                        String type_expected = ST.getParamType(i+1, func_name);
                        if(!arg_type.equals(type_expected)) {
                            System.out.println("ERROR: " + arg + " is of type " +
                             arg_type + " expected type of " + type_expected);
                        }
                    }
                    else {
                        System.out.println("ERROR: " + arg + " is not declared
                        in this scope");
                    }
                }
            }
        }
    }
```

If the arguments type does not match the actual expected type, an error message is written. An error message is also shown if the given argument is not declared within the scope.

Is every function called?

In order to do this check, I've created a HashSet to keep track of all the functions invoked. A HashSet will prevent any duplicates. After the Prog has finished, I can check if the functions contained in the global scope are all contained in the invoked functions HashSet, if not, an error message is printed with the function name which has not been invoked. To do this, I also created a helper method in the Symbol Table to return all the functions in it, by checking the type to be of "function".

```
public ArrayList<String> functionsToList() {
    LinkedList<String> list = symbolTable.get("global");
    ArrayList<String> functions = new ArrayList<String>();
    for(int i = 0; i < list.size(); i++) {
        if(!list.get(i).equals("global") && !list.get(i).equals("main")) {
            String description = descriptionTable.get(list.get(i)+"global");
            if(description.equals("function"))
                functions.add(list.get(i));
        }
    }
    return functions;
}
```

```
private void checkInvokedFunctions() {
  ArrayList<String> functions = ST.functionsToList();
  for(int i = 0; i < functions.size(); i++) {
      if(!invokedFunctions.contains(functions.get(i))) {
          System.out.println("ERROR: " + functions.get(i) + " is never invoked");
      }
  }
}
```

Intermediate Code Generation

The intermediate code representation was implemented similarly to the semantic analysis. I created a CodeGenerator that implements the CCALVisitor class. Again same as semantic analysis, it walks the AST and performs some operations for the code generation.

Using a combination of the lecture notes and some references made to this:
https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/13/Slides13.pdf
I implemented the following structure to the IR code.

Variables and Constants
Variables and constants are only printed if they are global in scope (as with the MIPS language). Variables are simply initiated while constants are defined.

```
VAR     i
CONST   i       =       5
```

```
// has two children, ID and TYPE
public Object visit(VarDec node, Object data) {
  String id = (String)node.jjtGetChild(0).jjtAccept(this, data);
  String type = (String) node.jjtGetChild(1).jjtAccept(this, data);
  if(node.jjtGetParent().toString().equals("Prog")) {
      System.out.println("VAR\t" + id);
  }
  return data;


public Object visit(ConstDec node, Object data) {
  String id = (String)node.jjtGetChild(0).jjtAccept(this, data);
  String type = (String) node.jjtGetChild(1).jjtAccept(this, data);
  String num = (String) node.jjtGetChild(2).jjtAccept(this, data);
  if(node.jjtGetParent().toString().equals("Prog")) {
      System.out.println("CONST\t" + id + "\t=\t" + num);
  }
  return data;
}
```

In the above snippets, you can see that variable and constant declarations only get printed when their parent is Prog which means they are global in scope.

Functions

Once a FUNCTION node is encountered, a label is created with that function name. The label is then labelled again with “begin” and “end” labels respectively. All of the FUNCTION node children get accepted one by one which will cause them to be printed out in sequence. These will be statements and they are handled separately. If a FUNCTION node has a child names FunctionReturn, it means that it has a value to return which is appended to the “ret” value to be returned at the end. I also calculate the number of bytes allocated for the parameters by getting the Parameter node and multiplying all of its children by 4 bytes respectively. When the function returns, the number of bytes are popped off.

This is an example of a function output. 8 bytes are popped as that function had 2 parameters. Statements and return is displayed. {i, y} is pushed onto the stack as it's getting ready to call a function so we push the parameters onto the stack and go to the function ("func") with the parameters of i and y.

Statements

Statement generation was some of the trickier parts of the IR code generation.

A simple statement of "x = 3" etc is done fairly easily by just accepting the first child of the statement as the left hand side of the statement and accepting the second child as the value. It can be printed as follows:

```
else {
    String value = (String) node.jjtGetChild(next).jjtAccept(this, data);

    if(id != null && value != null) {
        System.out.println("\t\t" + id + "\t= " + value);
    }
}
```

If a statement encountered a child of ArgList, it means that a statement is doing a function call. Here is where we identify the parameters we need to push onto the stack to be used before actually going to the function.

```
TEST_FN:
    begin
        i    = 2
        x    = 6
        push     i
        push     y
        goto     func
```

```
else if(childNode.equals("ArgList")) {
    int children = node.jjtGetChild(next).jjtGetNumChildren();
    for(int i = 0; i < children; i++) {
        String param = (String)node.jjtGetChild(next).jjtGetChild(i).
        jjtAccept(this, data);
        System.out.println("\t\tpush\t" + param);
    }
   System.out.println("\t\tgoto\t" +  id);
}
```

If a statement is an assignment to a function, we need to push the parameters onto the stack and then call the function by supplying the number of parameters to be used.

```
if(num > 0) {
    String childNode = node.jjtGetChild(next).toString();
    String id = (String) node.jjtGetChild(next-1).jjtAccept(this, data);
    if(childNode.equals("FuncReturn")) {
        int n = node.jjtGetChild(next).jjtGetNumChildren();
        if(n > 0) {
            String func_name = (String) node.jjtGetChild(next).jjtAccept(this,
            data);

            int children = node.jjtGetChild(next).jjtGetChild(next-1).
            jjtGetNumChildren();
            Node child = node.jjtGetChild(next).jjtGetChild(next-1);
            int param_count = 0;
            for(int i = 0; i < children; i++)
            {
                String param = (String) child.jjtGetChild(i).jjtAccept(this, data
                );
                System.out.println("\t\tparam\t" + param);
                param_count++;
            }
            System.out.println("\t\t" + id + "\t=\tcall " + func_name + ", " +
            param_count);
        }
        else {
            // must be an operation
            printOperation(node, data);
        }
    }
```

If on entering a statement, we find a child of FuncReturn, it can mean one of two things, it's either a function call to another function or it has an assignment with an operation (ie. multiple values being added/subtracted etc). If FuncReturn has multiple children, it is a function call. If it's a function call, we traverse through it's parameters and push each parameter on the stack by printing "param x". We keep count of the parameters and take note of the function name. When we are done with the parameters, we call the function with the function name and parameter count.

If the FuncReturn node has no children, then it is just one of a series of nodes from the parent statement. In order to traverse and deal with these children we pass the parent node (ie.Statement) to a helper function called "printOperation".

```
private void printOperation(Statement node, Object data) {
  String id = (String) node.jjtGetChild(0).jjtAccept(this, data);
  // last child will be Assign
  String result = id + " = ";
  for(int i = 1; i < node.jjtGetNumChildren() - 1; i++)
  {
      result += " " + node.jjtGetChild(i).jjtAccept(this, data);
  }
  System.out.println("\t\t" + result);
}
```

The first node of the Statement node here will always be the left hand side of the assignment which we call ID. We can accept this to get the name of the ID and append to our result. We traverse through the children will which be in the order of id/num etc followed by an operation (minus/plus) followed by another number of numbers/id's etc. We append each child here to the result which will provide us with an end result of "x = x + 3 - y" for example.

If/While statements

Statement which are "if/while" have an associated node.value with them which helps me identify potential if and why statements as these should be traversed a little differently.
When visiting a statement node, I check to see if it's node.value is not null and if it is either of "if" or "while". If it is, I know i will receive a combination of FuncReturn, Comparison, Condition nodes to deal with. FuncReturn nodes will give me all of the identifiers associated with the statement which the comparison will give me the appropriate comparison (less than, greater than etc) and the conditions will provide with conditions of AND or OR.
I traverse through each child, looking for either comparison, ids or conditions and add them to their respective arrays for later. When the statements children have been traversed, I can output my result by traversing through the ID, COMPS and CONDS arrays. I have done it this way as I need to traverse them in a different order than they are visited by the AST.

```
if(node.value != null) {
    if(node.value.equals("if") || node.value.equals("while")) {
    ArrayList<String> ids = new ArrayList<>();
    ArrayList<String> comps = new ArrayList<>();
    ArrayList<String> conds = new ArrayList<>();
    for(int i = 0; i < node.jjtGetNumChildren(); i++) {
        String n = node.jjtGetChild(i).toString();
        if(n.equals("FuncReturn")) {
            ids.add((String)node.jjtGetChild(i).jjtAccept(this, data));
            next++;
        }
        else if(n.equals("Comparison")) {
            String value = (String)node.jjtGetChild(i).jjtGetChild(0).jjtAccept(
            this, data);
            String comp = (String)node.jjtGetChild(i).jjtGetChild(1).jjtAccept(
            this, data);
            String comparison =  comp + " " + value;
            comps.add(comparison);
            next++;
        }
        else if(n.equals("ANDCondition") || n.equals("ORCondition")) {
            conds.add((String)node.jjtGetChild(i).jjtAccept(this, data));
            next++;
        }
    }
```

As i traverse through these statements, I keep track of the number of nodes or children I have gone through as the parent statement will most likely have much more children and this way I can index to them later. This is done with the "next" variable.
As the result gets printed out it gets assigned a label with a global variable of labelNumber which is incremented upon leaving the if/ while statement.

```
String result = "";
for(int i = 0; i < ids.size(); i++)
{
    result += ids.get(i) + " ";
    if(comps.size() > i) {
        result += comps.get(i);
    }
    if(conds.size() > i) {
        // condition statements are amended in reverse order so we traverse
        from the end
        result +=  " " +  conds.get(conds.size()-i-1) + " ";
    }
}
System.out.println("\t" + node.value + "\t" + "(" + result.trim() + ")" + "
goto label" + labelNumber);
System.out.println("\tlabel" + labelNumber + ":");
labelNumber++;
```

However, there are several problems with the IR code. As if/while statements get nested, these statements get missed so only parent if/while statements are transformed into IR code. I think if I had implemented the AST in a different manner to handle “else” statements and nested if statements it would have been an easier fix.

The IR code is outputted into its respective file with the “.ir” extension once the CodeGenerator has started.