**INSTITUTO TECNOLÓGICO DE COSTA RICA**

**ESCUELA DE INGENIERÍA EN COMPUTACIÓN**

**COMPILADORES E INTÉRPRETES**

**TERCER TAREA PROGRAMADA**

**GENERACIÓN DE CÓDIGO**

**PROFESOR: OSCAR VÍQUEZ ACUÑA**

**ESTUDIANTES: BRANDON CHAVES VILLEGAS**

**XAVIER BLANCO ARIAS**

**FECHA DE ENTREGA: 14/11/2014**

**Soluciones e implementación**

|  |  |
| --- | --- |
| Implementación de las condiciones: If. | Visit(context.condition());  Label l1 = globalIL.DefineLabel();  globalIL.Emit(OpCodes.Brfalse, l1);  Visit(context.statement(0));  globalIL.MarkLabel(l1);  if (context.TokenElse() != null)  {  Visit(context.statement(1));  }  return null;  Visita los nodos correspondientes del árbol y dentro de ellos chequea si se cumple la condición solicitada, en caso de no ser así realiza un salto de los procesos que se encuentran dentro de la condición.  Visita condTerm y además de eso analiza los resultados cuando se realiza el operador booleano “or”, si todas estas operaciones son verdaderas se agregará a la pila un 1 para representar que todas las comparaciones se cumplieron, de lo contrario se agrega un 0.  Visita condFact y además de eso analiza los resultados que se obtienen cuando se realiza el operador booleano “and”, si todas estas operaciones son verdaderas se agregará a la pila un 1 para representar que las comparaciones se cumplieron, de lo contrario se agrega un 0.  Recibe 2 expresiones y un comparador que las comparará entre ellas y de dicha comparación se obtendrá un valor booleano que se agrega a la pila. Retorna el comparador que recibió par que cuando sea llamado por su nodo padre este conozca el tipo de comparación que se realizó. |
| Implementación de declaración de métodos. | public override object VisitTypeFormVarMethodDeclAST([NotNull] ParserProgra.TypeFormVarMethodDeclASTContext context)  {  if (context.formPars() != null)  {  List<string> lista = new List<string>();  lista = (List<string>)Visit(context.formPars());  if (lista.Count > 0)  {  Type[] arrayType = new Type[lista.Count];  for (int i = 0; i < lista.Count; i++)  {  if (lista.ElementAt(i).Equals("int"))  {  arrayType[i] = typeof(int);  }  else if (lista.ElementAt(i).Equals("int[]"))  {  arrayType[i] = typeof(int[]);  }  else if (lista.ElementAt(i).Equals("char"))  {  arrayType[i] = typeof(char);  }  else if (lista.ElementAt(i).Equals("char[]"))  {  arrayType[i] = typeof(char[]);  }  else if (lista.ElementAt(i).Equals("boolean"))  {  arrayType[i] = typeof(Boolean);  }  else  {  TypeBuilder tipo = buscarClase(lista.ElementAt(i));  arrayType[i] = tipo;  }  }  string tipo2 = "";  if (context.type() != null)  {  tipo2 = (string)Visit(context.type());  if (tipo2.Equals("int"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(int),  arrayType); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else if (tipo2.Equals("int[]"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(int[]),  arrayType); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else if (tipo2.Equals("char"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(char),  arrayType); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else if (tipo2.Equals("char[]"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(char[]),  arrayType); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else if (tipo2.Equals("boolean"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(Boolean),  arrayType); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else  {  TypeBuilder tipo = buscarClase(tipo2);  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  tipo,  arrayType); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  }  else if (context.TokenVoid() != null)  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(void),  arrayType); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  }  }  else  {  string tipo2 = "";  if (context.type() != null)  {  tipo2 = (string)Visit(context.type());  if (tipo2.Equals("int"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(int),  null); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else if (tipo2.Equals("int[]"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(int[]),  null); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else if (tipo2.Equals("char"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(char),  null); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else if (tipo2.Equals("char[]"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(char[]),  null); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else if (tipo2.Equals("boolean"))  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(Boolean),  null); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  else  {  TypeBuilder tipo = buscarClase(tipo2);  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  tipo,  null); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  }  else if (context.TokenVoid() != null)  {  MethodBuilder methodBldr = globalMainType.DefineMethod(context.ID().GetText(),  MethodAttributes.Public |  MethodAttributes.Static,  typeof(void),  null); // Parametros de la funcion  globalIL = methodBldr.GetILGenerator();  methodArray.Add(methodBldr);  globalActualMethod = methodBldr;  }  }  if(context.varDecl()==null)  {  List<string> listaLocales = new List<string>();  methodFields.Add(listaLocales);  List<LocalBuilder> listalocales2 = new List<LocalBuilder>();  methodLocals.Add(listalocales2);    }  else  {  for (int i = 0; i < context.varDecl().Count; i++)  {  Visit(context.varDecl(i));  }  List<string> lista2 = new List<string>();  lista2.AddRange(ListaLocales);  methodFields.Add(lista2);  ListaLocales.Clear();  List<LocalBuilder> lista3 = new List<LocalBuilder>();  lista3.AddRange(ListaLocalesMetodo);  methodLocals.Add(lista3);  ListaLocalesMetodo.Clear();  }    Visit(context.block());  return null;  }  Se almacena el nombre del método declarado, variables que sean definidos en el cuerpo del método y sus parámetros dentro de listas globales por aparte. Visita type para saber el tipo de retorno del método, formPars que representa los parámetros que recibe, además visita varDecl y block que forman parte del cuerpo. |
| Implementación de factor. | Si es método realiza la llamada a dicho método, sino guarda a la pila el valor de la variable  Almacena el valor del número en la pila.  Almacena el valor del caracter en la pila.  Almacena el valor del booleano en la pila.  Visita a expression. |
| Implementación del ciclo “for”. | public override object VisitStatementForAST([NotNull] ParserProgra.StatementForASTContext context)  {  Visit(context.expr());  Label l1 = globalIL.DefineLabel();  if (context.conditionL != null)  {  Visit(context.condition());  if (context.statement(0) != null)  {  Visit(context.statement(0));  }  globalIL.Emit(OpCodes.Brfalse, l1);  }  if (context.statement(1) != null)  {  Visit(context.statement(1));  }  globalIL.MarkLabel(l1);  return null;  }  Visita condition y en caso de que la condición solicitada no se cumpla, el compilador se saltará lo que esté dentro del cuerpo del ciclo, fueron las indicaciones dadas pero debido a un error que no pudo ser controlado la implementación quedó incompleta. |
| Implementación del ciclo “while” | Visit(context.condition());  Label l1 = globalIL.DefineLabel();  globalIL.Emit(OpCodes.Brfalse, l1);  Visit(context.statement());  if (context.statement().GetText().Equals("break;"))  {  }  globalIL.MarkLabel(l1);  return null;  Visita condition y en caso de que la condición solicitada no se cumpla, el compilador se saltará lo que esté dentro del cuerpo del ciclo, fueron las indicaciones dadas pero debido a un error que no pudo ser controlado la implementación quedó incompleta. |
|  |  |

**Resultados Obtenidos**

El compilador es capaz de realizar efectivamente declaración de métodos, arreglos, variables y constantes. Es importante destacar que también permite el uso de índices para acceder a posiciones en los arreglos. Permite el retorno de valores en los métodos y el uso condiciones dentro de él.

En la declaración de clases permite el uso de sus atributos para realizar distintos procesos y obtener resultados deseados. Además las operaciones de suma, resta, multiplicación, división y división modular. Dichas operaciones pueden ser utilizadas vara asignar valores a variables, índices, ciclos, etc…

El proyecto no cuenta con la implementación completa de los ciclos pero si realiza comparaciones de

**Conclusiones**

El proyecto se puede considerar de alta dificultad ya que se debe tener conocimientos sobre la herramienta y debido a la amplitud que posee se dificulta conocer y utilizar de manera correcta todos los comandos que brinda.

**Bibliografía**

[**http://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes(v=vs.110).aspx**](http://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes(v=vs.110).aspx)

[**https://github.com/sagifogel/NJection.LambdaConverter/blob/master/NJection.LambdaConverter/DynamicProxies/ILEmitters/BinaryExpressionEmitter.cs**](https://github.com/sagifogel/NJection.LambdaConverter/blob/master/NJection.LambdaConverter/DynamicProxies/ILEmitters/BinaryExpressionEmitter.cs)

[**http://www.drdobbs.com/generating-code-at-run-time-with-reflect/184416570**](http://www.drdobbs.com/generating-code-at-run-time-with-reflect/184416570)