CSCE 4101 - Compiler Design
Semantic Analyzer and Code Generator for $C-$

Bishoy Boshra Labib   Nada Kamel Abdelhady

Dec. $1^{\text{st}}, 2016$

Contents

1 Attribute Grammar

The following grammar shows all the semantic rules needed for the tasks of:

I. **Semantic Analysis**, including symbol table construction, type checking and error handling

II. **Code Generation**

1.1 Program & Declarations

1. $program \rightarrow type\ identifier\ (parameters\ )\ "\{"\ declaration\_list\ compound\_statement\ "\}"$

   i. $program.\mathbf{code} = compound\_statement.\mathbf{code}$

2. $declaration\_list \rightarrow declaration\_list\ variable\_declaration$

3. $declaration\_list \rightarrow variable\_declaration$

4. $variable\_declaration \rightarrow type\ identifier\ ;$

   i. symbol_table.put($type.\mathbf{data\_type}$, $identifier$.lexim, simple, 0)

5. $variable\_declaration \rightarrow type\ identifier\ "["\ integer\_literal\ "]"\ ;$

   i. symbol_table.put($type.\mathbf{data\_type}$, $identifier$.lexim, array, $integer\_literal.\mathbf{value}$)

6. $type \rightarrow$ int

   i. $type.\mathbf{data\_type}$ = integer

   $type \rightarrow$ void

   i. $type.\mathbf{data\_type}$ = void

   $type \rightarrow$ float

   i. $type.\mathbf{data\_type}$ = real

7. $parameters \rightarrow parameter\_list$

8. $parameters \rightarrow$ void

9. $parameter\_list \rightarrow parameter\_list\ ,\ parameter$

10. $parameter\_list \rightarrow parameter$

11. $parameter \rightarrow type\ identifier$

i. symbol_table.insert(*type*.**data_type**, *identifier*.leximname, simple)

12. *parameter* → *type* *identifier* "[ ]"

    i. symbol_table.insert(*type*.**data_type**, *identifier*.lexim, array)

13. *compound_statement* → "{" *statement_list* "}"

    i. *compound_statement*.**code** = *statement_list*.**code**

14. *statement_list* $_1$ → *statement_list* $_2$ *statement*

    i. *statement_list* $_1$.**code** = *statement_list* $_2$ + *statement*.**code**

15. *statement_list* → *empty*

    i. *statement_list* $_1$.**code** = empty

16. *statement* → *compound_statement*

    i. *statement*.**code** = *compound_statement*.**code**

17. *statement* → *assignment_statement*

    i. *statement*.**code** = *assignment_statement*.**code**

18. *statement* → *selection_statement*

    i. *statement*.**code** = *selection_statement*.**code**

19. *statement* → *iteration_statement*

    i. *statement*.**code** = *iteration_statement*.**code**

1.2 Variables, Expressions & Assignment

1. *assignment_statement* → *variable* = *expression*

    i. *assignment_statement*.**data_type** = (*variable*.**data_type** == *expression*.**data_type**) ? *variable*.**data_type** : error

    ii. a. assignment_instruction = new Instruction(assign, *variable*.**address**, *expression*.**value**)

        b. *assignment_statement*.**code** = *variable*.**code** + *expression*.**code** + assignment_instruction

2. *variable* → *identifier* [1]

[1]While the variable address attribute is statically bound in this rule, it's dynamically bound in the following one

i. *variable*.**data_type** = symbol_table.lookup(*identifier*.lexim).**data_type**

ii. *variable*.**address** = symbol_table.lookup(*identifier*.lexim).**address**

iii. *variable*.**code** = empty

3. *variable* → *identifier* "[" *expression* "]"

i. *variable*.**data_type** = symbol_table.lookup(*identifier*.lexim).**data_type**

ii. symbol = symbol_table.lookup(*identifier*.lexim)

iii. *variable*.**base_address** = symbol.**address**

iv. *variable*.**address** = *variable*.**base_address** + *expression*.**value** * symbol.**data_type**.entity_size

v. a. offset = *expression*.**value**
   b. multiply = new Instruction(mul, offset, offset, symbol.**data_type**.size)
   c. add = new Instruction(add, *variable*.**address**, *variable*.**base_address**, offset)
   d. *variable*.**code** = *expression*.**code** + multiply + add

4. $expression_1 \rightarrow$ $expression_1$ *relational_operator* *addition_expression* [2]

i. *expression*.**data_type** = integer

ii. a. relational_instruction = new Instruction(*relation_operator*.op, $expression_1$.**value**, $expression_2$.**value**, *addition_expression*.**value**)
   b. $expression_1$.**code** = $expression_2$.**code** + *addition_expression*.**code** + relational_instruction

5. *expression* → *addition_expression*

i. *expression*.**data_type** = *addition_expression*.**data_type**

ii. *expression*.**value** = *addition_expression*.**value**

iii. *expression*.**code** = *addition_expression*.**code**

6. *relational_operator* → <

i. *relational_operator*.op = '<'

7. *relational_operator* → <=

i. *relational_operator*.op = '<='

8. *relational_operator* → >

i. *relational_operator*.op = '>'

[2] For simplicity, the use of temporary variables to compute `value` attributes is not reflected in this grammar

9. *relational_operator* $\rightarrow$ >=

   i. *relational_operator*.op = '>='

10. *relational_operator* $\rightarrow$ ==

   i. *relational_operator*.op = '=='

11. *relational_operator* $\rightarrow$! =

   i. *relational_operator*.op = '! ='

12. *addition_expression* $_1$ $\rightarrow$ *addition_expression* $_2$ *addition_operator* *term*

   i. *addition_expression*$_1$.**data_type** = (*addition_expression*$_2$.**data_type** == error || *term*.**data_type** == error || *addition_expression*$_2$.**data_type** != *term*.**data_type**) ? error : *term*.**data_type**

   ii. *addition_expression*$_1$.**value** = *addition_expression*$_2$.**value** *addition_operator*.op *term*.**value**

   iii. a. addition_instruction = new Instruction(*addition_operator*.op, *addition_expression*$_1$.**value**, *addition_expression*$_2$.**value**, *term*.**value**)

      b. *addition_expression*.**code** = *addition_expression*$_2$.**code** + *term*.**code** + addition_instruction

13. *addition_expression* $\rightarrow$ *term*

   i. *addition_expression*.**data_type** = *term*.**data_type**

   ii. *addition_expression*.**value** = *term*.**value**

   iii. *addition_expression*.**code** = *term*.**code**

14. *addition_operator* $\rightarrow$ +

   i. *addition_operator*.op = '+'

15. *addition_operator* $\rightarrow$ $-$

   i. *addition_operator*.op = '$-$'

16. *term*$_1$ $\rightarrow$ *term*$_2$ *multiplication_operator* *factor*

   i. *term* $_1$.**data_type** = (*term* $_2$.**data_type** != *factor*.**data_type** || *term* $_2$.**data_type** == error || *factor*.**data_type** == error) ? error : *factor*.**data_type**

   ii. a. multiply_instruction = new Instruction(*multiplication_operator*.op, *term* $_1$.**value**, *term* $_2$.**value**, *factor*.**value**)

      b. *term*$_1$.**code** = *term*$_2$.**code** + *factor*.**code** + multiply_instruction

17. *term* → *factor*

    i. *term*.**data_type** = *factor*.**data_type**
    ii. *term*.**value** = *factor*.**value**
    iii. *term*.**code** = *factor*.**code**

18. *multiplication_operator* → ∗

    i. *multiplication_operator*.op = '∗'

19. *multiplication_operator* → /

    i. *multiplication_operator*.op = '/'

20. *factor* → (*expression*)

    i. *factor*.**data_type** = *expression*.**data_type**
    ii. *factor*.**value** = *expression*.**value**
    iii. *factor*.**code** = *expression*.**code**

21. *factor* → *variable*

    i. *factor*.**data_type** = *variable*.**data_type**
    ii. a. load_instruction = new Instruction(Load, *variable*.**address**, *factor*.**value**)
        b. *factor*.**code** = *variable*.**code** + load_instruction

22. *factor* → *integer_literal*

    i. *factor*.**data_type** = integer
    ii. *factor*.**value** = *integer_literal*.**value**

23. *factor* → *real_literal*

    i. *factor*.**data_type** = real
    ii. *factor*.**value** = *real_literal*.**value**

1.3 Control Statements

1. *selection_statement* → if (*expression*) *statement*

    i. if (*expression*.**data_type** ! = integer) error_type(*selection_statement*)
    ii. a. exit_if_label = new_label()
        b. exit_if_jump = new instruction(jump_z, *expression*.**value**, exit_if_label)

c. *selection_statement*.**code** = *expression*.**code** + exit_if_jump + *statement*.**code** + exit_if_label

2. *selection_statement* → if (*expression*) *statement* $_1$ else *statement*$_2$

   i. if (*expression*.**data_type** ! = integer) error_type(*selection_statement*)

   ii. a. exit_if_label = new_label()
      b. exit_else_label = new_label()
      c. exit_if_jump = new instruction(jump_z, *expression*.**value**, exit_if_label)
      d. exit_else_jump = new instruction(jump, exit_else_label)
      e. *selection_statement*.**code** = *expression*.**code** + exit_if_jump + *statement*$_1$.**code** + exit_else_jump + exit_if_label + *statement*$_2$.**code** + exit_else_label

3. *iteration_statement* → while(*expression*) *statement*

   i. a. enter_while_label = new_label()
      b. enter_while_jump = new instruction(jump, enter_while_label)
      c. exit_while_label = new_label()
      d. exit_while_jump = new instruction(jump_z, *expression*.**value**, exit_while_label)
      e. *iteration_statement*.**code** = enter_while_label + *expression*.**code** + exit_while_jump + *statement*.**code** + enter_while_jump

2 Symbol Table Structure

The symbol table was implemented as a hash table, mapping a variable name to a variable object containing the following fields:

1. identifier
2. type
3. category (simple or array)
4. array size (in case of array)
5. relative memory address
6. declaration line number

This structure allowed the execution of the following tasks:

1. check if a variable has been defined (semantic analysis)
2. check for multiple definitions of a variable of the same name (semantic analysis)

3. check variable type (semantic analysis)
4. retrieve memory address of a variable (code generation)
5. retrieve type of a variable (code generation)

You can find the implementation of the symbol table in 4.5, and the implementation of the variable name class with the hashing function in 4.2

3 Instruction Set Architecture

The following are the instructions used in the generated code[3]

1. `Jump L`
   unconditionally jump to label L
2. `JumpZ rs L`
   jump to Label L, provided that $rs = 0$
3. `Assign rs rt`
   assign $rs = rt$
4. `Add rs rt rd`
   assign $rs = rt + rd$
5. `Sub rs rt rd`
   assign $rs = rt - rd$
6. `Mul rs rt rd`
   assign $rs = rt * rd$
7. `Div rs rt rd`
   assign $rs = rt/rd$
8. `SetL rs rt rd`
   assign $rs = \ rt < rd\ ?1\ :0$
9. `SetLE rs rt rd`
   assign $rs = \ rt \leq rd\ ?1\ :0$
10. `SetG rs rt rd`
    assign $rs = \ rt > rd\ ?1\ :0$

[3]A lot of these instructions are most likely to be implemented only as pseudo-instructions, especially in a RISC architecture. Notice also that the code might contain labels, which are only markers for certain addresses and not instructions that get executed.

11. `SetGE rs rt rd`
    $assign\ rs = \ rt \geq rd\ ?1\ : 0$

12. `SetE rs rt rd`
    $assign\ rs = \ rt == rd\ ?1\ : 0$

13. `SetNE rs rt rd`
    $assign\ rs = \ rt != rd\ ?1\ : 0$

14. `Load rs rt`
    $assign\ rt = Memory[rs]$

15. `Store rs rt`
    $assign\ Memory[rs] = rt$

4 Source Code

4.1 DataType

An enumeration for data types of our variables and constants

```
public enum DataType
{
		integer, real;
		public int getSize() throws Exception
		{
			switch(this)
			{
			case integer: return 2;
			case real: return 4;
			default: throw new Exception("DataType: " + this.name() +
					" has unrecognized size");
			}
		}
}
```

4.2 VariableName

The sole purpose of this class was to wrap a string (representing a variable name) in an object, where we can implement our own hashing function.

```
public class VariableName {
	String Name;
```

```java

        final static int MOD = 1000003; // a large prime
        final static int BASE = 128;

        VariableName(String name) {
                this.Name = name;
        }

        @Override
        public int hashCode() {
                int ans = 0;
                for (char c : this.Name.toCharArray())
                        ans = (ans * BASE + c) % MOD;
                return ans;
        }

        @Override
        public boolean equals(Object other) {
                return
                        other != null
                        && VariableName.class.isInstance(other)
                        && this.Name.equals(((VariableName) other).Name);
        }
}
```

4.3 VariableCategory

An enumeration for variable category

```java
public enum VariableCategory
{
        simple, array;
}
```

4.4 Variable

```java
public class Variable {
        // fields
        private String Name;
        private DataType Type;
        private int DeclarationLineNumber;

```

```
	private VariableCategory Category;
	private int ArraySize;
	private int RelativeAddress;

	// constructors
	Variable(int relativeAddress, Node identifier, Node type, Node arraySize)
			throws Exception {
		// set name
		this.Name = identifier.getLexim();
		// set declaration line number
		this.DeclarationLineNumber = identifier.getDeclarationLineNumber();
		// set type
		switch (type.getType()) {
		case int_keyword:
			this.Type = DataType.integer;
			break;
		case float_keyword:
			this.Type = DataType.real;
			break;
		default:
			throw new Exception("unrecognized type token: " + type.getLexim());
		}
		// set relative address
		this.RelativeAddress = relativeAddress;
		// set category and array size
		if (arraySize == null) {
			this.Category = VariableCategory.simple;
			this.ArraySize = 0;
		} else {
			this.Category = VariableCategory.array;
			this.ArraySize = Integer.parseInt(arraySize.getLexim());
		}
	}

	// getters
	public String getName() {
		return this.Name;
	}

	public boolean isArray() {
		return this.getCategory() == VariableCategory.array;
	}
```

```java

	public VariableCategory getCategory() {
		return this.Category;
	}

	public DataType getType() {
		return this.Type;
	}

	public int getArraySize() {
		return this.ArraySize;
	}

	public int getRelativeAddress() {
		return this.RelativeAddress;
	}

	public int getDeclarationLineNumber() {
		return this.DeclarationLineNumber;
	}
	public int getSize() throws Exception {
		return
			this.getType().getSize()
			* (this.isArray() ? this.getArraySize() : 1);
	}
}
```

4.5 SymbolTable

```java
import java.util.HashMap;

public class SymbolTable {
	private HashMap<VariableName, Variable> Table
		= new HashMap<VariableName, Variable>();
	private int DataSize = 0;

	private final static String PrintFormat = "%-10s%-10s%-10s%-10s%-10s%-10s";

	/*
	 * The constructor of the symbol table is passed a node of the syntax
	 * representing the declaration list
	 *
```

```
         * The constructor stores all variables in the hash table, so they will be
         * available for further retrieval
         */
        SymbolTable(Node declarationList) throws Exception {
                for (Node declaration = declarationList; declaration != null;
                                declaration = declaration.getSibling()) {
                        this.declare(declaration,
                                        declaration.getChild(0),
                                        declaration.getChildrenCount() == 2 ?
                                                        declaration.getChild(1)
                                                        : null);
                }
        }

        /*
         * return a variable defined by this name
         */
        public Variable lookUp(String name) throws Exception {
                if (!this.isDefined(name))
                        throw new Exception("Undefined variable: " + name);
                else return this.Table.get(new VariableName(name));
        }

        /*
         * tell whether or not the symbol table contains a variable with this name
         */
        public boolean isDefined(String name) {
                return this.Table.containsKey(new VariableName(name));
        }

        /*
         * print the content of the symbol table in a readable format
         */
        public void printContent() {
                System.out.println(String.format(PrintFormat, "line", "address",
                                "type", "name", "category", "array size"));
                for (Variable v : this.Table.values())
                        System.out.println(
                                String.format(PrintFormat,
                                        v.getDeclarationLineNumber(),
                                        v.getRelativeAddress(),
                                        v.getType(),
```

```
                                        v.getName(),
                                        v.getCategory(),
                                        v.getArraySize()));
        }

        private void declare(Node identifier, Node type, Node arraySize)
                        throws Exception {
                String name = identifier.getLexim();
                if (this.isDefined(name)) {
                        throw new Exception(
                                String.format(
                                        "redefinition of %s in line %d;" +
                                        "already defined in line %d;",
                                        name, identifier.getDeclarationLineNumber(),
                                        this.Table.get(new VariableName(name))
                                        .getDeclarationLineNumber()));
                } else {
                        Variable v = new Variable(DataSize, identifier, type, arraySize);
                        this.Table.put(new VariableName(name), v);
                        DataSize += v.getSize();
                }
        }
}
```

4.6 SemanticAnalyzer

```
public class SemanticAnalyzer {
        SymbolTable Table;
        Node StatementList;

        /*
         * The constructor of the semantic analyzer is passed a reference to the
         * symbol table and one to the node representing the statement list of the
         * syntax tree generated by the parse
         */
        public SemanticAnalyzer(SymbolTable table, Node statementList) {
                this.Table = table;
                this.StatementList = statementList;
        }

        /*
         * check if there are any semantic errors
```

```java
     */
    public void checkTypeErrors() throws Exception {
        checkStatement(this.StatementList);
    }

    /*
     * The following function loops over the statements in the statement list
     * passed, and checks for semantic errors
     */
    private void checkStatement(Node statement) throws Exception {
        for (; statement != null; statement = statement.getSibling()) {
            switch (statement.getType()) {
            case if_keyword: {
                getExpressionType(statement.getChild(0));
                checkStatement(statement.getChild(1));
                if (statement.getChildrenCount() == 3)
                    checkStatement(statement.getChild(2));
                break;
            }
            case while_keyword: {
                getExpressionType(statement.getChild(0));
                checkStatement(statement.getChild(1));
                break;
            }
            case assignment_operator: {
                // check for any type mismatch
                getExpressionType(statement);
                break;
            }
            }
        }
    }

    /*
     * the following function is passed an expression and returns its type
     *
     * It checks for the following errors:
     *
     * 1. mismatching of the types of the two operands of any operator
     * (assignment, addition, multiplication, relational)
     *
     * 2. use of any undeclared variables
```

```
	 */
	private DataType getExpressionType(Node node) throws Exception {
		switch (node.getType()) {
		case integer_literal:
			return DataType.integer;
		case real_literal:
			return DataType.real;
		case assignment_operator:
		case addition_operator:
		case relational_operator:
		case multiplication_operator: {
			DataType leftType = getExpressionType(node.getChild(0));
			DataType rightType = getExpressionType(node.getChild(1));
			if (leftType != rightType)
				throw new Exception(
					"type mismatch between operands of "
					+ node.getLexim() + " on line "
					+ Integer.toString(
						node.getDeclarationLineNumber()));
			else
				return leftType;
		}
		case identifier:
			return this.Table.lookUp(node.getLexim()).getType();
		case square_bracket:
			return getExpressionType(node.getChild(0));
		default:
			throw new Exception("unexpected token " + node.getLexim());
		}
	}
}
```

4.7 InstructionType

```
public enum InstructionType {
	Jump,
	JumpZ,

	Assign,

	Label,

```

```
        Add,
        Sub,
        Mul,
        Div,

        SetL,
        SetLE,
        SetG,
        SetGE,
        SetE,
        SetNE,

        Load,
        Store;
}
```

4.8 Instrucion

```
public class Instruction {
        /*
         * This class represents a single instruction in the three-address code
         * generated by the code generator It contains a type and up to 3 operands
         */
        private InstructionType Type;
        private String Operand[] = new String[3];

        /*
         * constructor is passed the instruction type, and the operands
         */
        Instruction(InstructionType type, String op0) {
                this(type, op0, null, null);
        }

        Instruction(InstructionType type, String op0, String op1) {
                this(type, op0, op1, null);
        }

        Instruction(InstructionType type, String op0, String op1, String op2) {
                this.Type = type;
                this.Operand[0] = op0;
                this.Operand[1] = op1;
                this.Operand[2] = op2;
```

```
        }

        public InstructionType getType() {
                return this.Type;
        }
        public String getOperand(final int idx)
        {
                return this.Operand[idx];
        }
        /*
         * print the instruction on a line in a readable format
         */
        public void print() {
                if (this.getType() == InstructionType.Label) {
                        System.out.println(this.Operand[0] + ":");
                } else {
                        System.out.print(String.format("%-10s", this.Type.name()));
                        System.out.print(String.format("%-5s", this.Operand[0]));
                        if (this.Operand[1] != null)
                                System.out.print(String.format("%-5s", this.Operand[1]));
                        if (this.Operand[2] != null)
                                System.out.print(String.format("%-5s", this.Operand[2]));
                        System.out.println();
                }
        }
}
```

4.9 CodeGenerator

Notice that I have also implemented a scheme that keeps track of the number of temporary registers used at a moment, thus the generated code usually does not use temporary register with large indeces. However, there exists a much more efficient scheme that time did not permit to implement.

```
import java.util.*;

public class CodeGenerator {
        int labelCounter;
        Node StatementList;
        SymbolTable Table;
        ArrayList<Instruction> code;

```

```java
	/*
	 * The constructor of the code generator is passed a reference to the symbol
	 * table, and a reference to the statement list of the program
	 */
	public CodeGenerator(SymbolTable table, Node statementList) {
		this.Table = table;
		this.StatementList = statementList;
	}

	/*
	 * return the code generated by traversing the statement list that was
	 * passed to the constructor
	 */
	public ArrayList<Instruction> generateCode() throws Exception {
		labelCounter = 0;
		code = new ArrayList<Instruction>();
		this.generateStatementCode(this.StatementList);
		return code;
	}

	private void generateStatementCode(Node statement) throws Exception {
		for (; statement != null; statement = statement.Sibling) {
			switch (statement.getType()) {
			case if_keyword: {
				/*
				 * generate code to evaluate the expression and
				 * store the result in t0
				 */
				int expressionRegNum = 0;
				generateExpressionCode(statement.getChild(0),
						expressionRegNum);
				Instruction exitIfLabel = newLabel();
				Instruction exitIf = new Instruction(
						InstructionType.JumpZ,
						tmpReg(expressionRegNum),
						exitIfLabel.getOperand(0));
				/*
				 * add a conditional jump to exit label
				 */
				code.add(exitIf);
				/*
				 * generate code for statements that are supposed
```

```
                     * to be executed when the expression evaluates
                     * to true
                     */
                    generateStatementCode(statement.getChild(1));
                    if (statement.getChildrenCount() == 3) {
                            /*
                             * if the if statement has an else part,
                             * generate code for it
                             */
                            Instruction exitElseLabel = newLabel();
                            /*
                             * add a jump instruction (to skip the exit
                             * part in case the expression evaluated to
                             * true and the previous part was executed
                             */
                            Instruction exitElse = new Instruction(
                                            InstructionType.Jump,
                                            exitElseLabel.getOperand(0));
                            code.add(exitElse);
                            /*
                             * place the 'if' exit label here (so the code
                             * will execute the else part if the expression
                             * evaluates to false and the jump is executed
                             */
                            code.add(exitIfLabel);
                            /*
                             * generate code for the else part
                             */
                            generateStatementCode(statement.getChild(2));
                            /*
                             * place the 'else' exit label here
                             */
                            code.add(exitElseLabel);
                    } else {
                            /*
                             * if there is no else part, we just need to
                             * just place the 'if' exit label here
                             */
                            code.add(exitIfLabel);
                    }
                    break;
            }
```

```
                case while_keyword: {
                        /*
                          * code for iteration statement uses labels in a way
                          * similar to that for selection statement
                          */
                        Instruction enterLabel = newLabel();
                        Instruction exitLabel = newLabel();
                        code.add(enterLabel);
                        int expressionRegNum = 0;
                        generateExpressionCode(statement.getChild(0),
                                        expressionRegNum);
                        Instruction exitWhile = new Instruction(
                                        InstructionType.JumpZ,
                                        tmpReg(expressionRegNum),
                                        exitLabel.getOperand(0));
                        code.add(exitWhile);
                        generateStatementCode(statement.getChild(1));
                        Instruction enterWhile = new Instruction(
                                        InstructionType.Jump,
                                        enterLabel.getOperand(0));
                        code.add(enterWhile);
                        code.add(exitLabel);
                        break;
                }
                case assignment_operator: {
                        /* generate code to evaluate expression on right hand
                          * side and store it in register #valueRegNum
                          */
                        int valueRegNum = 0;
                        generateExpressionCode(statement.getChild(1),
                                        valueRegNum);
                        Node variable = statement.getChild(0);
                        /* the following code finds the memory location
                          * of the variable on the left hand side, and
                          * adds a 'store' instruction to store the value
                          * computed previously in that location
                          */
                        if (variable.getType() == TokenType.square_bracket) {
                                int addressRegNum = 1;
                                int offsetRegNum = 2;
                                Node id = variable.getChild(0);
                                Node expression = variable.getChild(1);
```

```java
                                        /*
                                          * generate code to compute the expression
                                          * representing the array index, and store
                                          * its value in register #offsetRegNum
                                          */
                                        generateExpressionCode(expression, offsetRegNum);
                                        /*
                                          * generate code to assign value of register
                                          * #addressRegNum to the base address of the
                                          * variable
                                          *
                                          * base address is a static attribute that
                                          * is obtained from the symbol table
                                          */
                                        Instruction assign = new Instruction(
                                                        InstructionType.Assign,
                                                        tmpReg(addressRegNum),
                                                        address(id));
                                        /*
                                          * generate code to multiply offset by
                                          * the size of an array element of this
                                          * type
                                          *
                                          * size of an array element is a static
                                          * attribute that is obtained from the
                                          * symbol table
                                          */
                                        Instruction mul = new Instruction(
                                                        InstructionType.Mul,
                                                        tmpReg(offsetRegNum),
                                                        tmpReg(offsetRegNum),
                                                        size(id)
                                                        );
                                        Instruction add = new Instruction(
                                                        InstructionType.Add,
                                                        tmpReg(addressRegNum),
                                                        tmpReg(addressRegNum),
                                                        tmpReg(offsetRegNum));
                                        Instruction store = new Instruction(
                                                        InstructionType.Store,
                                                        tmpReg(addressRegNum),
                                                        tmpReg(valueRegNum));
```

```java
                                code.add(assign);
                                code.add(mul);
                                code.add(add);
                                code.add(store);
                        } else {
                                Instruction store = new Instruction(
                                                InstructionType.Store,
                                                address(variable),
                                                tmpReg(valueRegNum));
                                code.add(store);
                        }
                        break;
                }
                }
        }
}

/*
 * this function generates code to be evaluate the expression
 * represented by the passed syntax tree node, and stores its
 * value in temporary register #returnRegNum
 */
private void generateExpressionCode(Node node, int returnRegNum)
                throws Exception {
        switch (node.getType()) {
        case addition_operator:
        case multiplication_operator:
        case relational_operator: {
                int leftValueRegNum = returnRegNum + 1;
                int rightValueRegNum = returnRegNum + 2;
                // if the expression is an operator, generate code
                // over the following 3 steps:
                // 1. generate code to store the value of the left child in
                // register #leftValueRegNum
                generateExpressionCode(node.getChild(0), leftValueRegNum);
                // 2. generate code to store the value of the right child in
                // register #rightValueRegNum
                generateExpressionCode(node.getChild(1), rightValueRegNum);
                // 3. add an instruction to combine the result of the left
                // child and right child; first find the operator type, and
                // then add the instruction
                InstructionType t = InstructionType.Add;
```

```
                switch (node.getLexim()) {
                case "+":
                        t = InstructionType.Add;
                        break;
                case "-":
                        t = InstructionType.Sub;
                        break;
                case "*":
                        t = InstructionType.Mul;
                        break;
                case "/":
                        t = InstructionType.Div;
                        break;
                case "<=":
                        t = InstructionType.SetLE;
                        break;
                case ">=":
                        t = InstructionType.SetGE;
                        break;
                case "==":
                        t = InstructionType.SetE;
                        break;
                case "!=":
                        t = InstructionType.SetNE;
                        break;
                case "<":
                        t = InstructionType.SetL;
                        break;
                case ">":
                        t = InstructionType.SetG;
                        break;
                }
                Instruction operation = new Instruction(t,
                                tmpReg(returnRegNum),
                                tmpReg(leftValueRegNum),
                                tmpReg(rightValueRegNum));
                code.add(operation);
                break;
        }
        case integer_literal:
        case real_literal: {
                // if the expression is just a constant, directly
```

```
                // store its value in the register
                Instruction assign = new Instruction(
                                InstructionType.Assign,
                                tmpReg(returnRegNum),
                                node.getLexim());
                code.add(assign);
                break;
        }
        case identifier: {
                // if the expression is a variable, load its value
                // from memory to the register
                Instruction load = new Instruction(InstructionType.Load,
                                address(node),
                                tmpReg(returnRegNum));
                code.add(load);
                break;
        }
        case square_bracket: {
                int addressRegNum = returnRegNum + 1;
                int offsetRegNum = returnRegNum + 2;
                // if the expression is an array element, compute
                // its address first and then load the value from
                // memory into the return register
                generateExpressionCode(node.getChild(1),
                                offsetRegNum);
                Instruction assign = new Instruction(InstructionType.Assign,
                                tmpReg(addressRegNum),
                                address(node.getChild(0)));
                Instruction mul = new Instruction(InstructionType.Mul,
                                tmpReg(offsetRegNum),
                                tmpReg(offsetRegNum),
                                size(node.getChild(0)));
                Instruction add = new Instruction(InstructionType.Add,
                                tmpReg(addressRegNum),
                                tmpReg(addressRegNum),
                                tmpReg(offsetRegNum));
                Instruction load = new Instruction(InstructionType.Load,
                                tmpReg(addressRegNum),
                                tmpReg(returnRegNum));
                code.add(assign);// assign base address address
                code.add(mul);        // multiply offset by element size
                code.add(add);        // increment address
```

```
                        code.add(load);         // load from memory
                        break;
                }
                }
        }
        /*
         * the following function creates a new label
         */
        private Instruction newLabel()
        {
                return new Instruction(InstructionType.Label,
                                "L" + Integer.toString(labelCounter++));
        }
        /*
         * I have only used the following function to eliminate pieces of code that
         * are repeated multiple times. This code just returns a string representing
         * some type of operands of the instructions
         */
        private String address(Node id) throws Exception {
                // passed a node representing a variable, return its address
                return Integer.toString(this.Table.lookUp(id.getLexim())
                                .getRelativeAddress());
        }
        private String size(Node id) throws Exception {
                // passed a node representing a variable, return its entity size
                return Integer.toString(this.Table.lookUp(id.getLexim())
                                .getType().getSize());
        }

        private String tmpReg(final int idx) {
                // return a string representing a temporary register
                return "t" + Integer.toString(idx);
        }
}
```

5 Test Cases

I. The program was tested on the following input:

```
int main(void)
{
```

```
        int x;
        int y[5];
        {
                x = (5 + 3) * (x + 7);
                y[3] = x;
                {
                        x = 5;
                        x = 5;
                }
                while(x == 5)
                        if (y[0] == 6)
                                x = 7 + x;
                        else { x = 7 - x; }
        }
}
```

The output of the parser was as follows, showing that it worked correctly.[4]

```
line      address   type      name      category  array size
4         2         integer   y         array     5
3         0         integer   x         simple    0
Assign    t2   5         't2 contains 5
Assign    t3   3         't3 contains 3
Add       t1   t2   t3   't1 contains 3 + 5
Load      0    t3        't3 contains content of memory location 0 (address
                         'of x) thus, t3 contains value of x
Assign    t4   7         't4 contains 7
Add       t2   t3   t4   't2 contains x + 7
Mul       t0   t1   t2   't0 contains (3 + 5) * (x + 7)
Store     0    t0        'store content of t0 into memory location 0 (address
                         'of x)
Load      0   t0         't0 contains content of x
Assign    t2   3         't2 contains array index
Assign    t1   2         't1 contains address of y
Mul       t2   t2   2    't2 contains offset
Add       t1   t1   t2   't1 contains address of y[3]
Store     t1   t0        'store value of t0(x) into address t1 (y[3])
Assign    t0   5
Store     0    t0        'store value of t0 into memory location 0 (x)
```

[4] I have manually commented some the instructions in this example, to help explain further the instruction set architecture used. Notice also that we do not show the output of the scanner and the parser for conciseness, since at this stage, we are only interested in the work of the semantic analyzer and the code generator.

```
Assign    t0   5
Store     0    t0
L0:                      'label used to repeat the while loop
Load      0    t1        't1 contains value of x
Assign    t2   5         't2 contains 5
SetE      t0   t1   t2   'compare t1 and t2, and set t0 accordingly
JumpZ     t0   L1        'if t0 is zero, exit while loop
Assign    t3   0         't3 contains array index
Assign    t2   2         't2 contains address of y
Mul       t3   t3   2    't3 contains offset
Add       t2   t2   t3   't2 contains address of y[0]
Load      t2   t1        't1 contains content of y[0]
Assign    t2   6         't2 contains 6
SetE      t0   t1   t2   'compare t1(y[0]) and t2(6), and set t0 accordingly
JumpZ     t0   L2        'jump to the else part, if the expression is false
Assign    t1   7         't1 = 7
Load      0    t2        't2 = x (because memory_address(x) = 0)
Add       t0   t1   t2   't0 = 7 + x
Store     0    t0        'store it into x
Jump      L3             'skip the else part of the if-statement
L2:                      'label used to skip if part and jump to else part
Assign    t1   7         't1 = 7
Load      0    t2        't2 = value of x
Sub       t0   t1   t2   't0 = t1 - t2 = 7 - x
Store     0    t0        'x = t0
L3:                      'label used to skip the else part
Jump      L0             'go to the start of the while loop
L1:                      'label used to exit the while loop
```

II. The program was tested on the following input:

```
int main(void)
{
        int x;
        int y;
        int z;
        int a[3];
        {
                a[0] = 1;
                a[1] = 1;
                a[2] = 1;
                x = 3 * a[0] + 4 * a[1] + 5 * a[2];
```

```
                if (z == x)
                {
                        while(x == z)
                        {
                                z = y * 2;
                                x = z + 1;
                        }
                }
                else
                {
                        while(x != z)
                        {
                                x = a[0] * 2;
                                a[0] = 2 * a[1] + 3 * y + 5 * (x + y + z);
                        }
                }
        }
}
```

The output of the parser was as follows, showing that it worked correctly.

```
line      address   type      name      category  array size
6         6         integer   a         array     3
5         4         integer   z         simple    0
4         2         integer   y         simple    0
3         0         integer   x         simple    0
Assign    t0   1
Assign    t2   0
Assign    t1   6
Mul       t2   t2   2
Add       t1   t1   t2
Store     t1   t0
Assign    t0   1
Assign    t2   1
Assign    t1   6
Mul       t2   t2   2
Add       t1   t1   t2
Store     t1   t0
Assign    t0   1
Assign    t2   2
Assign    t1   6
Mul       t2   t2   2
```

```
Add       t1  t1  t2
Store     t1  t0
Assign    t3  3
Assign    t6  0
Assign    t5  6
Mul       t6  t6  2
Add       t5  t5  t6
Load      t5  t4
Mul       t2  t3  t4
Assign    t4  4
Assign    t7  1
Assign    t6  6
Mul       t7  t7  2
Add       t6  t6  t7
Load      t6  t5
Mul       t3  t4  t5
Add       t1  t2  t3
Assign    t3  5
Assign    t6  2
Assign    t5  6
Mul       t6  t6  2
Add       t5  t5  t6
Load      t5  t4
Mul       t2  t3  t4
Add       t0  t1  t2
Store     0   t0
Load      4   t1
Load      0   t2
SetE      t0  t1  t2
JumpZ     t0  L0
L1:
Load      0   t1
Load      4   t2
SetE      t0  t1  t2
JumpZ     t0  L2
Load      2   t1
Assign    t2  2
Mul       t0  t1  t2
Store     4   t0
Load      4   t1
Assign    t2  1
Add       t0  t1  t2
```

```
Store     0    t0
Jump      L1
L2:
Jump      L3
L0:
L4:
Load      0    t1
Load      4    t2
SetNE     t0   t1   t2
JumpZ     t0   L5
Assign    t3   0
Assign    t2   6
Mul       t3   t3   2
Add       t2   t2   t3
Load      t2   t1
Assign    t2   2
Mul       t0   t1   t2
Store     0    t0
Assign    t3   2
Assign    t6   1
Assign    t5   6
Mul       t6   t6   2
Add       t5   t5   t6
Load      t5   t4
Mul       t2   t3   t4
Assign    t4   3
Load      2    t5
Mul       t3   t4   t5
Add       t1   t2   t3
Assign    t3   5
Load      0    t6
Load      2    t7
Add       t5   t6   t7
Load      4    t6
Add       t4   t5   t6
Mul       t2   t3   t4
Add       t0   t1   t2
Assign    t2   0
Assign    t1   6
Mul       t2   t2   2
Add       t1   t1   t2
Store     t1   t0
```

```
Jump      L4
L5:
L3:
```

III. The program was tested on the following code, where the variable x is define twice.

```
1  int main(void)
2  {
3          int x;
4          int x[5];
5          {
6                  x = (5 + 3) * (x + 7);
7                  y[3] = x;
8                  {
9                          x = 5;
10                         x = 5;
11                 }
12                 while(x == 5)
13                         if (y[0] == 6)
14                                 x = 7 + x;
15                         else { x = 7 - x; }
16         }
17 }
```

The output of the parser was as follows, showing that it worked correctly.

```
redefinition of x in line 4; already defined in line 3;
```

IV. The program was tested on the following code, where the variable x is not defined before use.

```
int main(void)
{
        int y[5];
        {
                x = (5 + 3) * (x + 7);
                y[3] = x;
                {
                        x = 5;
                        x = 5;
                }
                while(x == 5)
```

```
                        if (y[0] == 6)
                                x = 7 + x;
                        else { x = 7 - x; }
        }
}
```

The output of the parser was as follows, showing that it worked correctly.

```
line      address   type      name      category  array size
3         0         integer   y         array     5
Undefined variable: x
```

V. The program was tested on the following code, where the variable x is assigned to y which has a different type.

```
int main(void)
{
        int x;
        float y;
        {
                x = y;
        }
}
```

The output of the parser was as follows, showing that it worked correctly.

```
line      address   type      name      category  array size
4         2         real      y         simple    0
3         0         integer   x         simple    0
type mismatch between operands of = on line 6
```