CODE GENERATION FOR JVM, .NET, AND MIPS TARGETS FROM C MINUS LANGUAGE

A Thesis

Submitted to the Department of Computer Science and Engineering

of

BRAC University

by

Md. Zahurul Islam

Student ID: 01101002

In Partial Fulfillment of the

Requirements for the Degree

of

Bachelor of Science in Computer Science

May 2005

DECLARATION

I hereby declare that this thesis is based on the results found by myself. Materials of work found by other researcher are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted for any degree.

Signature of
Supervisor

Signature of
Author

ACKNOWLEDGMENTS

I would like to thank my thesis advisor Dr. Mumit Khan for being so patient and supportive. He provided me with valuable suggestion and references to research which were vital to the completion of this thesis and who taught me to keep my pace in producing this thesis.

I would like to thank Dr. Sayeed Salam who gave me encouragement to do a good thesis work.

I must thank Mr. Moinul Islam zaber, who first made me realize that the Bachelor of Computer Science thesis is a serious and demanding project.

My warmest appreciate goes to all of my friends at BRAC University specially Bashirul Islam sohel and Mehedi Al Mamun, who gave me mental support in completing this thesis.

ABSTRACT

The C was one of the well-built and popular language in the field of computer science. Now this days programmers are move away form C to use in the field of software development. One of the main causes is most of the available C compilers generate machine dependent code.
In this thesis we implement compiler that generate Java byte code from subset of C language in addition also generate byte code for MS .NET Common Language Runtime. Java Virtual Machine can execute the generated byte code in any machine/platform. Our implemented compiler can also generate code for MIPS machines.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Background

The C language is a general-purpose programming language that was originally designed by Dennis Ritchie of Bell Laboratories and implemented there on a PDP-11 in 1972. It was first used as the systems language for the UNIX operating system. The developer of UNIX, Ken Thompson, had been using both assembler and a language named B to produce initial versions of UNIX in 1970.

The invention of C came about to overcome the limitations of B. B was a programming language based on BCPL. Martin Richards developed BCPL as a tape less systems programming language. BCPL basic data type was the machine word, and it made heavy use of pointers and address arithmetic. C evolved from B and BCPL and incorporated typing.

C is a powerful, flexible language that provides fast program execution and imposes few constraints on the programmer. It allows low-level access to information and commands while retaining the portability and syntax of a high-level language.

C includes bit-wise operators along with powerful pointer manipulation capabilities. C imposes few constraints on the programmer. Another strong point of C is its use of modularity. Sections of code can be stored in libraries for re-use in future programs. This concept of modularity also helps with C's portability and

execution speed. These qualities make it a useful language for both systems programming and general-purpose programs.

Today, various C compilers are available around us. All compilers generate executable code form C source code. These executable codes are machine/platform dependent. These codes are unable to execute in different platform/machine. To execute in different platform programmers need to recompile the source code again. This is the main disadvantage of available C compiler and extra burden for C programmer.

Our research focuses on implanting a compiler from subset of C language that reduce the extra burden of programmers and solve that dependency problem.

1.2 Related Work

There is few works done related our thesis. Most of these focused on multi targets code generation and code generation for Java Virtual Machine. In this section we will discuss about two major C compilers those are generate code for multi target machine and Java Virtual Machine.

1.2.1 Lcc: A Retargetable ANSI C Compiler

Lcc is widely used compiler for standard C described in A Retargetable C Compiler [19]. Lcc can be linked with code generators for several targets, so it can be used as either a native compiler or a cross-compiler. It is distributed with back ends for the SPARC, MIPS, X86, and ALPHA for a variety platform.

1.2.2 Axiomatic Multi-Platform C (AMPC)

AMPC [20] is a C compiler that generates Java Bytecode. The resulting executables will be able to run on any Java Virtual Machine® (JVM). AMPC

enables programmers to develop new applications using the C language targeting the JVM. One can also use AMPC to turn legacy applications written in C into platform independent applications running on any JVM-enabled device. AMPC is based on ANSI C (1989). Please note that although AMPC is not 100% ANSI C compliant, it covers a very large subset of it. A JNI (JVM Native Interface) feature is available for calling native C or C++ functions from AMPC.

1.3 Our Goals

Our main goal is to solve the platform/machine dependency problem of current available C compilers. In addition, another goal is implement a compiler that generates code for multi- target machine from subset of C language. In short a compiler, that generates MIPS assembly language, Byte-code for Java Virtual Machine and Byte-code for Common Language Runtime.

Chapter II

SPECIFICATION OF C- LANGUAGE

In this chapter, we discuss the feature of C- Language. Section 1 introduces the lexical convention of C- language. Section 2 discusses the grammar for the C- language. It indicates which features of C language are covered by the C-. at the last section 3 we give the C- language semantics.

2.1 Lexical Conventions of C– Language

The C- (Minus) Language, based on Kenneth Louden's Compiler Construction: Principle and Practice [1]. The C- is that it is C with data type integer, string without any string operation and an additional Boolean type. There is no pointer in C- language and without some control constructs, and without any records (i.e., structure).

The C- language supports some keywords, operators, special symbol, string literal, integer literal, identifier and single line comment.

Keywords

int, bool, string, void, true, false, if, else, while, return.

Operators

+ - * / < <= > >= == != = -(unary) !

Special Symbols

; , ( )[ ] { } //

Other Tokens

ID, INT_LITERAL, STR_LITERAL

An Identifier consist of a letter, followed by zero or more letters, digits, or underscores and these are case sensitive. An INT_LITERAL is an integer literal (a digit followed by zero or more digits). A STR_LITERAL is character string literal, surrounded by double quotes. White space consists of blanks, new lines, and tabs. White space must separate IDs, INT_LITERALs, STR_LITERALs and keywords. Comments start with //, and extends to the end of the line. The following regular expression describes the ID, INT_LITERAL, STR_LITERAL tokens:

letter = [a-zA-Z]
digit = [0-9]
ID = letter (letter | digit | '_')*
INT_LITERAL = digit digit* = digit+
STR_LITERAL = '"' [^"]* '"'

2.2 Grammar of the C- Language

We are following this grammar for the C- language:

1. program → declaration-list
2. declaration-list → declaration-list declaration | ε
3. declaration → var-declaration | fun-declaration
4. var-declaration → type-specifier ID ; | type-specifier ID [INT_LITERAL] ;
5. type-specifier → int | bool | string | void
6. fun-declaration → type-specifier ID (params) compound-stmt
7. params → param-list | void | ε
8. param-list → param-list , param | param
9. param → type-specifier ID | type-specifier ID []
10. compound-stmt → { local-declarations statement-list }

11. local-declarations → local-declarations var-declaration | ε
12. statement-list → statement-list statement | ε
13. statement → compound- stmt | assign-stmt | selection-stmt | iteration-stmt
    | call-stmt | return-stmt
14. selection-stmt→ if (expression) statement
    | if (expression) statement else statement
15. iteration-stmt → while (expression) statement
16. return-stmt → return ; | return expression ;
17. call-stmt → call ;
18. assign-stmt → var = expression ;
19. var → ID | ID [expression]
20. expression → expression mulop additive-expression
    | expression addop additive-expression
    | expression relop additive-expression
    |(expression)
    | var
    | call
    | INT_LITERAL
    | STR_LITERAL
    | true
    | false
21. relop → <= | < | > | >= | == | !=
22. addop → + | -
23. mulop → * | /
24. call → ID (args)
25. args → arg-list| ε
26. arg-list → arg-list , expression | expression

2.3 Language Semantics

The explanation of the associated semantics of C- Language is given below:

1. program → declaration-list
2. declaration-list → declaration-list declaration | ε
3. declaration → var-declaration | fun-declaration

A program consists of a list (or sequence) of declarations, which may be variable or function declarations, in any order. There may be zero declaration, which will be detected and rejected by semantic analysis because any legal C-program must have a main function (see below). Semantic restrictions are as follows (these do not occur in C). All variables and functions must be declared before they are used (this avoids backpatching references). The last declaration in a program must be a function declaration of the form void main (void). Note that C– lacks prototypes, so that no distinction is made between declarations and definitions (as in C).

4. var-declaration → type-specifier ID **;** | type-specifier ID [INT_LITERAL] ;
5. type-specifier → int | bool | string | void

A variable declaration declares either a simple variable of integer type or an array variable whose base type is integer and whose indices range from 0 ... INT_LITERAL-1. Note that in C– the basic types are int (integer)**,** bool (boolean or logical), string (character strings) and void. In a variable declaration, any type specifier other than void can be used. Void is for function declarations (see below). Note, also, that only one variable can be declared per declaration.

6. fun-declaration → type-specifier ID (params) compound-stmt
7. params → param-list | void

8. param-list → param-list **,** param | param
9. param → type-specifier ID | type-specifier ID [ ]

A function declaration consists of a return type specifier, an identifier, and a comma-separated list of parameters inside parentheses, followed by a compound statement with the code for the function. If the return type of the function is void, then the function returns no value (i.e., it is a procedure). Parameters of a function are either void or empty indicating that there are no parameters, or a list representing the function's parameters. Parameters followed by brackets are array parameters whose sizes can vary. Simple integer parameters are passed by value. Array parameters are passed by reference (i.e., as pointers) and must be matched by an array variable during a call. Note that there are no parameters of type function. The parameters of a function have scope equal to the compound statement of the function declaration, and each invocation of a function has a separate set of parameters. Functions may be recursive (to the extent that declaration before use allows).

10. compound-stmt → { local-declarations statement-list }

A compound statement consists of curly brackets surrounding a set of declarations and statements. A compound statement is executed by executing the statement sequence in the order given. The local declarations have scope equal to the statement list of the compound statement and supersede any global declaration.

11. local-declarations → local-declarations var-declaration | ε
12. statement-list → statement-list statement | ε

Note that both declaration and statement lists may be empty.

13. statement → compound-stmt
| assign-stmt
| selection-stmt
| iteration-stmt
| call-stmt
| return-stmt

There are six types of statements in a C— program.

14. selection-stmt→ if (expression) statement
| if (expression) statement else statement

The if-statement has the usual semantics: the logical expression (of type bool) is evaluated; a true value causes execution of the first statement; a false value causes the execution of the second statement, if it exists. This rule results in the classical dangling else ambiguity, which is resolved in the standard way: the else part is always parsed immediately as a substructure of the current if (the most closely nested disambiguating rule).

15. iteration-stmt → while **(** expression **)** statement

The while-statement is the only iteration statement in C–. It is executed by repeatedly evaluating the expression and then executing the statement if the expression evaluates to a false value, and ending when the expression evaluates to false.

16. return-stmt → return **;** | return expression **;**

A return statement may either return a value or not. Functions not declared void must return values. Functions declared void must not return values. A return causes transfer of control back to the caller (or termination of the program if it is inside main).

17. call-stmt → call **;**

A call-stmt is used for functions calls that do not return a value (i.e., with void return type), or whose return value is not assigned to a variable.

18. assign-stmt → var **=** expression
19. var → ID | ID [expression]

An assign-stmt assigns the value of the expression to the given variable. The assignment has the usual storage semantics: the location of the variable represented by var is found, then the sub expression to the right of the assignment is evaluated, and the value of the sub expression is stored at the given location. A var is either a simple (integer) variable or subscripted array variable. A negative subscript causes the program to halt (unlike C). However, upper bounds of the subscripts are not checked. Note that unlike in C, C— assignment statements are not expressions, so we cannot chain multiple assignments together (e.g., a = b = c = 5;). vars represent a further restriction of C– from C. In C, the target of an assignment must be an l-value, and l-values are addresses that can be obtained by many operations. In C–, the only l-values are those given by the var syntax, and so this category is checked syntactically, instead of during type checking is in C. Thus, pointer arithmetic is forbidden in C–

27. expression → expression mulop additive-expression
| expression addop additive-expression
| expression relop additive-expression

|(expression)
| var
| call
| INT_LITERAL
| STR_LITERAL
| true
| false

20. relop → <= | < | > | >= | == | !=
21. addop → + | -
22. mulop → * | /

Note that the grammar above does not specify the precedence and associativity of the operators, which must be corrected, either by using yacc's or Jcup's %left, right, %nonassoc, etc. directives or by re-writing the grammar. The relational operators or relops do not associate (that is, an unparenthesized expression can only have one relational operator). The / symbol represents integer division; that is, any remainder is truncated. An array variable must be subscripted, except in the case of an expression consisting of a single ID and used in a function call with an array parameter (see below).

23. call → ID (args)
24. args → arg-list | ε
25. arg-list → arg-list **,** expression | expression

A function call consists of an ID (the name of the function), followed by parentheses enclosing its arguments. Arguments are either empty or consist of a comma separated list of expressions, representing the values to be assigned to parameters during a call. Functions must be declared before they are called, and

the number of parameters in a declaration must equal the number of arguments in a call. An array parameter in a function declaration must be matched with an expression consisting of a single identifier representing an array variable.

Finally, the above rules give no input and output statements. We must include such functions in the definition of C–, since unlike C, C– has no separate compilation or linking facilities. We, therefore, consider the following functions to be predefined in the global environment, as though they had the indicated declarations:

```
int read_int (void);
string read_string (void);
void write_int (int x);
void write_string (string s);
```

The read_int function for example has no parameters and returns an integer value from the standard input device (usually the keyboard); and the write_int function takes an expression, which of course must evaluate to an integer value (i.e., cannot be a call to a function with void return type), whose value it prints to the standard output device (usually the screen), together with a newline. The other read and write functions do the same for the corresponding types.

Chapter III

COMPILER FRONT END

The Previous chapter discussed the C- language specification and the language semantics. Now we move on to presenting the front end of the C- compiler. Section 1 presents the lexical analysis. In section 2, we elaborate the syntax analysis and after that, semantic analysis comes at section 3.We break up the semantic analysis in two parts, name checking and type checking.

3.1 Lexical Analysis

Lexical analysis is the first phase of a compiler. This analysis performed by a lexical analyzer. The main task of lexical analyzer [14] is to read the input character and produce a sequence of tokens that the parser uses for syntax analysis. Lexical analyzer typically comments and white space in the form of blank, tab, and new line character. The sample C- minus program in Figure 3.1 calculates the “Greatest Common Divisor” and generated tokens from that program are given in Figure 3.2.

```
1:   // A program to perform Euclid's
2: // Algorithm to compute gcd.

3: int gcd (int u, int v)
4: {
5:   if ( v == 0) return u;
6:   else return gcd (v,u-u/v*v);
7: }
8: void main (void)
9: {
10:  int x; int y;
11:  x = 8;
```

```
12:  y = 12;
13:  // answer is 4
14:  write_int(gcd(x,y));
15: }
```

Figure 3.1 A sample C- program that calculate GCD from 12 and 8

```
Int     v       0      gcd     /     void     )      ;        12        ,
gcd     )       )       (      V     main     {      X        ;         y
 (      {    return     v      *       (     int     =    write_int     )
int     if       u      ,      V     void     x      8         (        )
 u      (        ;      u      )     main     ;      ;        gcd       ;
 ,      v     else      -      ;       (     int     Y         (        }
int     ==    return     u      }     void     y      =         x
```

Figure 3.2 generated tokens from program in figure 3.1

We use Jlex [14] scanner generator for lexical analysis. The Jlex produces java code from the source program. The input source to Jlex is a specification that includes a set of regular expressions and associated actions. The output of Jlex is a Java source file that defines a class named *Yylex*. Yylex includes a constructor that is called with one argument: the input stream (an InputStream or a Reader). It also includes a method called *next_token,* which returns the next token in the input. The code for lexical analysis is available in the Appendix. The List of tokens is shown in Table 3.1

3.2 Syntax Analysis

Syntax analysis is the second phase of a compiler. A parser performs this task. The parser is the hart of a typical compiler. The parser call the lexical analyzer to obtain the tokens of the input program, assembles the tokens together into a parse tree, and passes the tree to the later phase of the compiler, which perform semantic analysis and code improvement and generation. The position of syntax analysis in compiler model is given in figure 3.3.

Table: 3.1
Tokens and Tokens description

Token	Description
T_INT	Int keyword
T_BOOL	bool keyword
T_STRING	string keyword
T_VOID	Void keyword
T_TRUE	True keyword
T_FALSE	False keyword
T_IF	if keyword
T_ELSE	else keyword
T_WHILE	While keyword
T_RETURN	return keyword
T_AND	Logical and operator
T_OR	Logical or operator
T_NOT	Unary not operator
T_PLUS	Arithmetic addition operator
T_MINUS	Arithmetic subtraction operator
T_MULT	Arithmetic multiplication operator
T_DIV	Arithmetic division operator
T_LT	Relational less-than operator
T_LE	Relational less-than-or-equal operator
T_GT	Relational greater-than operator
T_GE	Relational greater-than-or-equal operator
T_EQ	Relational equal operator
T_NE	Relational not-equal operator
T_ASSIGN	Assignment symbol
T_SEMI	Semicolon symbol
T_COMMA	Comma operator
T_LPAREN	Left parenthesis
T_RPAREN	Right parenthesis
T_LBRACK	Left bracket
T_RBRACK	Right bracket
T_LBRACE	Left curly bracket
T_RBRACE	Right curly bracket
T_ID	identifier
T_INT_LITERAL	integer literal
T_STR_LITERAL	string literal
T_ERROR	Any error symbol

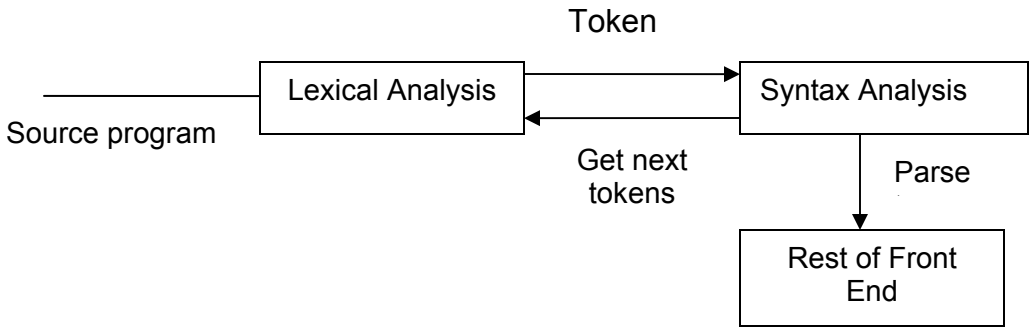

Figure 3.3 Position Syntax Analysis in front-end compiler model

To perform syntax analysis of the C- compiler we use the CUP (Java Based on Constructor of Useful Parser) parser [15]. CUP reads the grammar specification from the cup file and generates an LR (1) parser for it. The parser consist of a set of LALR (1) parsing tables and derive routine written in the java programming language. Using CUP involves creating a simple specification based on the grammar for which a parser is needed, along with construction of a scanner capable of breaking characters up into meaningful tokens (such as keywords, numbers, and special symbols). The syntax analyzer code is given in the Appendix.

In the syntax analysis phase, parser generates the parse tree. During parsing we convert the parse tree to the Abstract Syntax Tree (AST) [4].The internal nodes of an AST represent the operators and the children node represent the operands. By contrast, a parse tree is called a concrete syntax tree [4], and the underlying grammar is called a concrete syntax for the language. AST differs from parse tree because superficial distinctions of form, unimportant for translation, do not appear in syntax tree. Grammars for simple arithmetic, parse tree and AST for 8*4+2 are given in Figure 3.4. We build the `AST` in a class hierarchy. In the class hierarchy, the top class is the `AST`; program `AST`, `DeclListAST`, `DeclAST`, `StmtListAST`, `StmtAST`, `ExpListAST`, `ExpAST`, and `TypeAST` classes are inherited from `AST` class. The

`DeclListAST` contains list of declaration, the `StmtlistAST` contains list of statements, and the `ExpList` contains list of expression. The C- language support scalar variable, array variable, scalar and array parameter, and function declaration these are inherited from `DeclAST`. All statements are inherited from `StmtAST` and all expressions are inherited from `ExpAST`. Each binary and unary expressions are inherited from `ExpAST`. The C- language supports for data type integer, string, boolean, and void which are inherited from `TypeAST`. The AST hierarchy is given in figure 3.5.

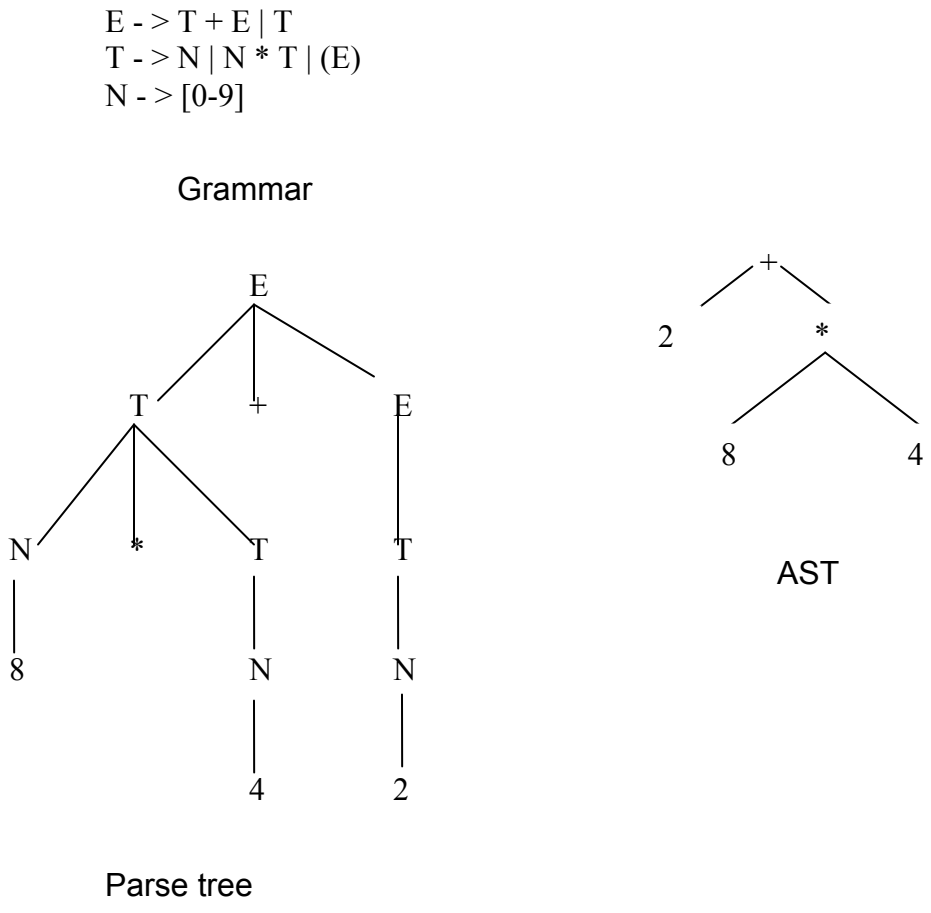

Figure 3.4 Simple Arithmetic Evaluation Grammar, Parse tree and AST 8*4+2

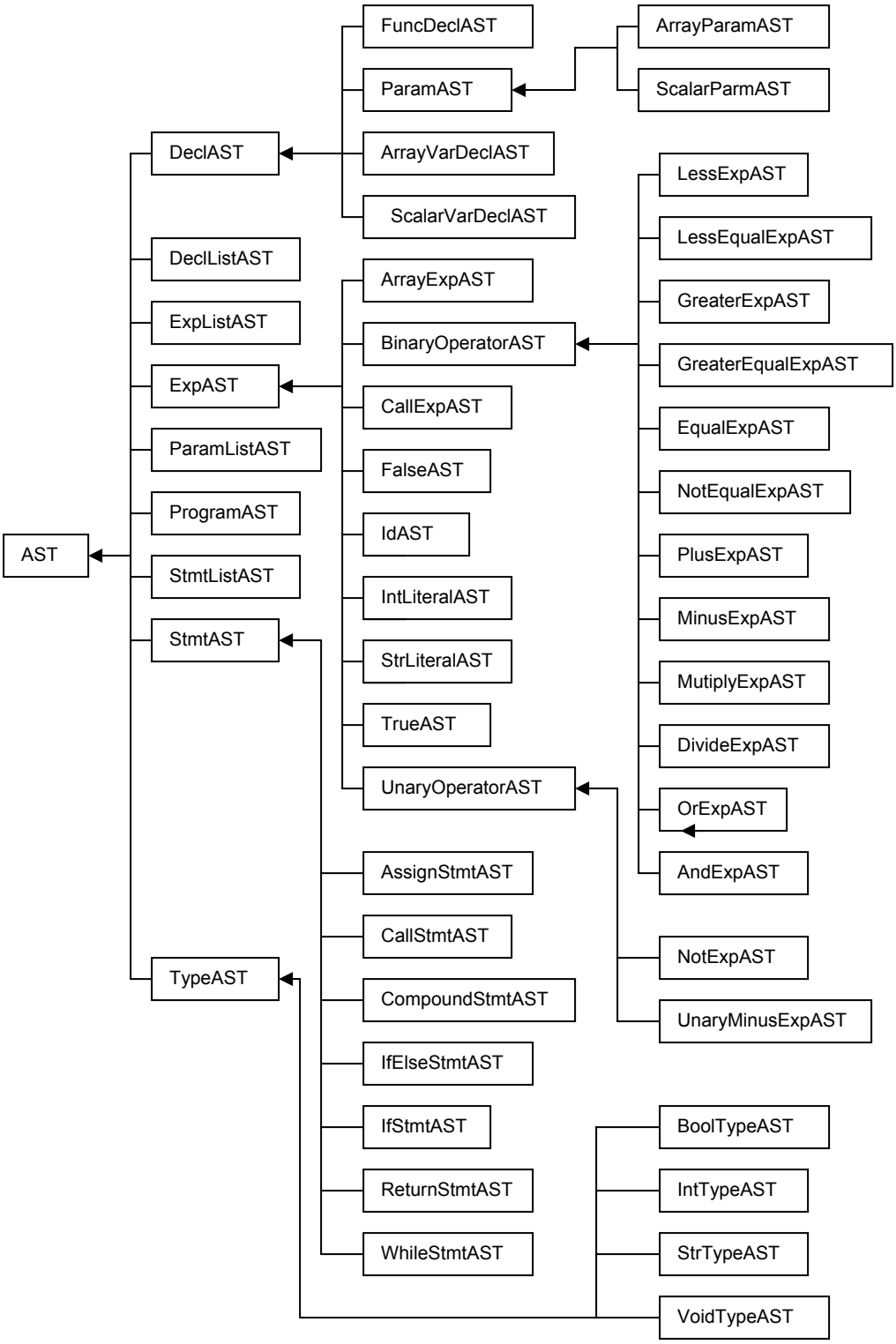

Figure 3.5 AST class hierarchies

3.4 Semantic Analysis

In previous two sections, we discussed about the language syntax. Now we move to semantic analysis. Formally, syntax concerns the form of a valid program, while semantics concern its meaning. It is conventional to say that the syntax of a language is precisely that portion of the language definition that can be described conventionally by a context-free grammar [4], while the semantics is that portion of the definition that cannot. When we require, for example, that the number of arguments contained in a call to a function match the number of formal parameter in the function definition, it is tempting to say that this requirement is a matter of syntax. After all, we can count arguments without knowing what they mean. Unfortunately, we cannot count them with context-free rules, or in the presence of separate compilation.

Semantics rules are further divide into static and dynamic semantics, though again the line between the two is somewhat fuzzy. The compiler enforces static semantics rules at compile time. It generates code to enforce dynamic semantic rules at run time. We follow the static semantics for the compiler. To analyze the semantics of a source program first insert the all symbol name, Symbol type and scope of that symbol in a symbol table. we break up the semantics analysis into two phases. First, one is name check and other one is type check.

3.4.1 Visitor

To perform semantic analysis and generate code for targets we use visitor pattern. The purpose of the Visitor Pattern [21] is to encapsulate an operation that performs on the elements of a data structure. In this way, we can change the operation being performed on a structure without the need of changing the classes of the elements that we are operating on. Using a Visitor

pattern, we allow decoupling the classes for the data structure and the algorithms used upon them.

Each node in the data structure "accepts" a Visitor, which sends a message to the Visitor, which includes the node's class. The visitor will then execute its algorithm for that element. This process is known as "Double Dispatching." The node makes a call to the Visitor, passing itself in, and the Visitor executes its algorithm on the node. In Double Dispatching, the call made depends upon the type of the Visitor and of the Host (data structure node), not just of one component.

The key is the Accept method in the Concrete Element classes. The body of this method shows the double dispatching call, where the Visitor is passed in to the accept method, and that visitor is told to execute its visit method, and is handed the node by the node itself. This makes for very robust code, since all of the decision making as to what to execute where and when it taken care of by the dispatching.

The Double Dispatching technique is that it entirely replaces conditional statements, making for much more robust code. Because the Host passes itself in to the visitor, the visitor knows where to execute the algorithm, and because of the type of the Host, it knows which method to run. This eliminates a lot of decision-making: normally, we have to decide (many times) at run-time what to do where, but here, all of the decision-making has been completed in the design stage. No object ever has to ask any questions; they just do what it is that they do. The polymorhic dispatching takes care of all of the decision-making [21].

One key advantage of using the Visitor Pattern is that adding new operations to perform upon data structure is very easy. All we have to do is create a new Visitor and define the operation there. This is the result of the very distinct separation of variant and invariant behavior in the Visitor pattern. The

data structure elements and the Visitor represent the invariant behaviors. The variant behaviors are encapsulated in the concrete Visitors.

Every Visitor has a method for every data structure element. The data structure elements however, only deal with the Visitor, and hence only have one method (accept ()) that deals with it. That method is overridden in each concrete element, which only calls its respective method in the Visitor.

3.4.2 Name Check

A name is mnemonic character string used to represent something else. Name in most language are identifier (alphanumeric token), though certain other symbols, such as + or -, also be names. Names allow us to refer to variables, constants, operation, types, and so on using symbolic identifiers rather than low-level concepts like address. In our C- language name are identifiers. Two important data abstraction for perform name check are symbol and symbol table [5].

3.4.2.1 Symbol

Each symbol contains name, type, and regardless scope in C- compiler. Three type symbol scope global, local and parameter are used in C-, global for global variable and function name, local for local variable and parameter for function parameter. All symbols are entered into a single large hash table, keyed by name [5].

3.4.2.2 Symbol Table

To keep track of the names in statically scoped program, a compiler relies on a data abstraction called s symbol table. At the basic level, the symbol table [5] is a dictionary: it maps names to the information the compiler knows about them. The most basic operation, which we call insert and find or lookup,

serve to place a new mapping (a name- to – object binding) into the table and to retrieve the information held in the mapping for a given name . As the semantic analysis phase of the C- compiler scans the code from beginning to end, it could insert new mapping at the beginning of each scope and remove them at the end. To manage the C- compiler we use global and local symbol table. At the beginning of name check, a global symbol table is created and stores the built in function, global variable, and user defined function. To find a symbol into symbol table we use find function that takes symbol name and scope. By checking the scope, the find function will call the local find or global find function. In find local function, we find the symbol only into the current symbol table and global find function first find into all available symbol tables but start from current. The global find function is given in Figure 3.6

```
1:  private Sym findGlobal(String name)
2:  {
3:   Iterator i = DictList.iterator();
4:   while (i.hasNext())
5:    {
6:      Hashtable temp = (Hashtable) i.next();
7:      Sym s = (Sym) temp.get(name);
8:      if (s != null)
9:      return s;
10:  }
11:  return null;
12: }
```

Figure 3.6 Global find function

3.4.2.3 Name checking of identifier

To perform the name check only need to check the name of identifiers. In all the class related to declaration into the C- compiler class hierarchy, the symbols are inserted into the symbol table. The name check of identifiers is performed in the IdAST class. We find the symbol for each identifier into symbol table, if not found prompt an error message. By name check user will be confirm

that there is no multiple declaration of any variable and function and each variable is declared.

3.4.3 Type Check

A compiler must check that the source program follows both the syntactic and semantic convention of the source language. This checking, called static checking, ensure that certain kinds of programming errors will be detected and reported. If name check of any source program is successful, then move to check type of that source program [6]. There are four data type int, bool, string, and void in the C- language.

3.4.3.1 Type checking of function

The application of a function to an argument can be captured by the production

E ->E (E)

in which an expression is the application of one expression to another. The reules for associating type expression with nonterminal T can be augmented by the following production and action to permit function type in declaration [4].

T -> T1' '->' T2 { T.type := T1.type ->T2.type }

Quotes around the arrow used as function constructor distinguish it form the arrow used as the meta symbol in a production.

The rule for checking the type of a function application is

E -> E1 (E2) {E.type := if E2.type = s and E1.type = S -> t then t
Else type_error}

This rule says that in an expression expression formed by applying E1 to E2, the type of E1 must be a function s -> t from the type s of E2 to some range type t. The type of E1(E2) is t [4].

3.4.3.2 Type checking of statements

Since language constructs like statement typically do not have values, the special basic type can be assigned to them. If an error is detected within a statement, the type assigned to the statement is type error. The condition of control statement (i.e. if statement, if else statement and while statement) must be Boolean. The translation scheme for checking the type of statements is given at Figure 3.7

```
S -> id: = E  {S.type:= if id.Type= E.type then void
                        else type_error}
S ->if E then S1 {S.type:= if E.type = Boolean then S1.type
                        else type_error}
S -> while E do S1 {S.type:= if E.type = Boolean then S1.type
                        else type_error}
S -> S1; S2       {S.type:= if S1.type = void and
                        S2.type= void then void
                        else type_error}
```

Figure 3.7 Translation scheme for checking the type of statements.

3.4.3.3 Type checking of Expression

In the type checking of expression, the following rules,, the synthesized attribute type for E gives the type expression assigned by the type system to the expression generated by E. The following semantic rules say that constants represented by the tokens STR_LITERAL and INT_LITERAL have type and integer, respectively:

E -> STR_LITERAL { E.type := string }
E ->INT_LITERAL { E.type := integer}

The expression formed by applying the unary logical operator the expression type is Boolean and resulting type also Boolean type. The rule is:

```
E -> unop E1          { E.type := if E1.type = Boolean the Boolean
                                  else type_ error}
```

The expression formed by applying the binary relational operator to two sub-expression of type integer has type Boolean; otherwise its type is type_error. The rule is:

```
E -> E1.biRelaOp E2   {E.type := if E1.type = integer and
                                 E2.type = integer then Boolean
                                 Else type_error }
```

The expression formed by applying the binary arithmetic operator to two sub-expression of type integer has type integer; otherwise, its type is type_error. The rule is:

```
E -> E1.biRelaOp E2   {E.type := if E1.type = integer and
                                 E2.type = integer then integer
                                 Else type_error }
```

In an array reference E1[E2], the index expression E2 must have type integer, in which case the result is the element type t obtain form the type array (s,t) of E1. We make no use of index set of the array.

```
E -> E1 [E2]          { E.type :=if E2.type = integer and
                                 E1.type = array (s,t) then t
                                 else type_error.}
```

Chapter IV

CODE GENERATION

In this chapter, we will give the description of compiler back end. Initially we only focus on code generation; code optimization will be our future development. In the first section, we will discuss about the basic of code generation. Next following section, we will discus about MIPS [7] code generation. The third section will be Oolong code generation for Java virtual machine [3]. In last section, we will discuss about the Microsoft intermediate language [18] and Byte-code generation for Common Language Runtime [8].

4.1 Code Generation Basics

The final phase in our compiler model is the code generation. It takes as input an intermediate representation of the source program and produces as output as equivalent target program. The target program code must be correct and high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.

We take MIPS [7] code generation as a base line. However, our goal is the generate code for multiple target machine. Our target machines are SPIM simulator, which take MIPS assembly language; second, one is the Java Virtual Machine and Common Language Runtime. We store all necessary things of a C-program into our Abstract Syntax tree class hierarchy and all the element in the hierarchy are well named and well typed. We will start code generation from the top of the Abstract Syntax tree, which is Programs. Intuitively we will visit all the

node of the class hierarchy. For three-type code generation we have used three visitors those will visit the class hierarchy.

We use pure stack machine to generate code for all targets. Stack machines are simple evaluation model. The stack contains the result of an operation. The invariant of stack machine is the before and after calling method the stack will remain same.

4.2 MIPS code Generation

Our base line code generation is to generate code for SPIM S20 simulator [17]. SPIM can read immediately executable files containing assembly language or MIPS [7] (Million Instruction per second) executable files. The MipsCodegenVisitor visit the Abstract Syntax tree class hierarchy and generate the MIPS assembly language. In the MIPS code generation

We use several MIPS register in generating code. The used MIPS registers name explanation are given in Table 4.1

Register	**Explanation**
$fp	Frame pointer
$sp	Stack pointer
$ra	Return address register
$ v0	Function/expression result
$v1	Function/expression result
$a0	Function argument
$t0	Temporary register
$t1	Temporary register

Table: 4.1 Used MIPS register and explanation

4.2.1 SPIM Simulator

SPIM S20[17] is a simulator that runs programs for the MIPS [7] R2000/R3000 RISC computers.The architecture of the MIPS computers is simple and regular, which makes it easy to learn and understand. The processor contains 32 general-purpose registers and a well-designed instruction set that make it a propitious target for generating code in a compiler. SPIM simulates two coprocessor. Coprocessor 0 handles exceptions, inturrupts, and the virtual memory system.SPIM similates most of the first two and entirely omits details of memory system. Coprocessor 1 is the floating- point unit. SPIM simulates most aspects of this unit.

Though workstations that contain a hardware, and hence significantly faster, implementation of this computer the simulators are become popular because these workstations are not generally available. Another reason is that these machine will not persist for many years because of the rapid progress leading to new and faster computers. Unfortunately, the trend is to make computers faster by executing several instructions concurrently, which makes their architecture more difficult to understand and program. The MIPS architecture may be the epitome of a simple, clean RISC machine [7].

In addition, simulators can provide a better environment for low-level programming than an actual machine because they can detect more errors and provide more features than an actual computer. For example, SPIM has a X-window interface that is better than most debuggers for the actual machines.

Finally, simulators are an useful tool for studying computers and the programs that run on them. Because they are implemented in software, not silicon, they can be easily modified to add new instructions, build new systems such as multiprocessors, or simply to collect data.

The MIPS architecture, like that of most RISC computers [20], is difficult to program directly because of its delayed branches, delayed loads, and restricted address modes. This difficulty is tolerable since these computers were designed to be programmed in high-level languages and so present an interface designed for compilers, not programmers. A good part of the complexity results from delayed instructions. A delayed branch takes two cycles to execute. In the second cycle, the instruction immediately following the branch executes. This instruction can perform useful work that normally would have been done before the branch or it can be a nop (no operation). Similarly, delayed lo*ads* take two cycles so the instruction immediately following a load cannot use the value loaded from memory.

MIPS wisely choose to hide this complexity by implementing a virtual machine with their assembler. This virtual computer appears to have non-delayed branches and loads and a richer instruction set than the actual hardware. The assembler reorganizes (rearranges) instructions to fill the delay slots. It also simulates the additional, pseudoinstructions by generating short sequences of actual instructions. By default, SPIM [17] simulates the richer, virtual machine. It can also simulate the actual hardware.

4.2.2 Overview of MIPS Assembly Language

MIPS is an load-store architecture, which means that only load and store instruction access memory. Computation instruction operates only on values in register. The bare machine provides only one memory-addressing mode: c (rx), which uses the sum of the immediate C and register Rx as the address. The virtual machine provides few addressing modes for load and store instruction. The MIPS addressing modes are given in Table 4.2

Most load and store instruction operate only on aligned data. A quantity is aligned if its memory address is a multiple of its size in bytes. Therefore, half-

word object must be stored at even address and a full word object must be stored at addresses that are a multiple of four.

Table 4.2
MIPS Addressing Modes

Format	Address Computation
(register)	Contents of register
Imm	Immediate
imm (register)	Immediate + contents of register
Label	Address of label
Label ± imm	Address of label + or – immediate
l Label ± imm (register)	Addess of label + or (immediate + contents of register)

MIPS syntaxes are pretty easy. Comments in assembler files begin with a sharp sign (#). Everything from the sharp sign to end of the line is ignored. Identifiers are a sequence of alphanumeric character, underscore (_), and dots (.) that do not begin with a number. Instruction opcode are reserved word that cannot be used as identifiers. Label are declared by putting them at the beginning of line followed by a colon.

4.3 Oolong Code Generation

One of our goals is generating Byte-code form C- language and Java Virtual Machine [2] [3] is our target machines. To generate the Byte-code [2] for Java Virtual machine we use Oolong [2] assembly language for Java virtual Machine as an intermediate language. We generate the Oolong code from the C– source code. To execute an Oolong program, it must be assembled into a class file. An Oolong assembler is available which assemble the Oolong language to class file. It is a free assembler developed by Joshua Engel. By convention, Oolong source file names end in .j. To generate Oolong assembly code we use several Oolong instructions. The used Oolong instructions and explanation are given in Table 4.3.

4.3.1 Java Virtual Machine

The Java virtual machine [2] is the cornerstone of the Java and Java 2 platforms. It is the component of the technology responsible for its hardware- and operating system- independence, the small size of its compiled code, and its ability to protect users from malicious programs.

The Java virtual machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time. It is reasonably common to implement a programming language using a virtual machine; the best-known virtual machine may be the P-Code machine of UCSD Pascal [3].

The first prototype implementation of the Java virtual machine, done at Sun Microsystems, Inc., emulated the Java virtual machine instruction set in software hosted by a handheld device that resembled a contemporary Personal Digital Assistant (PDA). Sun's current Java virtual machine implementations, components of its Java™ 2 SDK and Java™ 2 Runtime Environment [3] products, emulate the Java virtual machine on Win32 and Solaris hosts in much more sophisticated ways. However, the Java virtual machine does not assume any particular implementation technology, host hardware, or host operating system. It is not inherently interpreted, but can just as well be implemented by compiling its instruction set to that of a silicon CPU. It may also be implemented in microcode or directly in silicon.

The Java virtual machine knows nothing of the Java programming language, only of a particular binary format, the class file format. A class file contains Java virtual machine instructions (or byte codes) and a symbol table, as well as other ancillary information.

Table: 4.3
Used Oolong instructions and description

Instruction	Description
.class	Class directive name of the class
.super	Indicate super class of that class
.field	Field declaration (global variable in C-)
.method	Represent method new method declaration
inocst_n	Push n on the stack, 0 < n <= 5
bipush n	Push n on the stack, -128 <= n <= 127
sipush n	Push n on the stack, -32768 <= n <= 32767
newarray type	Allocate an array of given type.
Return	Control return to caller
ireturn	Control return to caller with int in the stack
Dup	Duplicate the top of the stack
Goto label	Control transfer to label
Invokestatic	Static invocation
Invokevirtual	Virtual invocation
ineg (stack: int1)	Negate the int1 (-int1)
If_icmplt label (stack: int1, int2)	If int1 < int2, control branches to label.
If_icmple label (stack: int1, int2)	If int1 <= int2, control branches to label.
If_icmpgt label (stack: int1, int2)	If int1 > int2, control branches to label.
If_icmpge label (stack: int1, int2)	If int1 >= int2, control branches to label.
If_icmpeq label (stack: int1, int2)	If int1 = int2, control branches to label.
If_icmpne label (stack: int1, int2)	If int1! = int2, control branches to label.
ifne label (stack: int1)	If int1! = 0, control branches to label.
ifeq label (stack: int1)	If int1 ==0, control branches to label.
iadd (stack: int1, int2)	Add int1 and int2 (int1 + int2)
isub (stack: int1, int2)	Subtract int2 from int1 (int1 – int2)
imul (stack: int1, int2)	Multiply int1 and int2 (int1 * int2)
idiv (stack: int1, int2)	Divide int1 by int2 (int1/int2)
ldc n	Push n on the stack
isotre n	Store an int in local variable n
astore n	Store a reference in local variable n
iload n	Load an int from local variable n
aload n	Load an reference from local variable n
Pop	Pop the top of the stack

For the sake of security, the Java virtual machine imposes strong format and structural constraints on the code in a class file. However, any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine. Attracted by a generally available, machine-independent platform, implementers of other languages are turning to the Java virtual machine as a delivery vehicle for their languages.

4.3.2 Overview of Oolong Language

Oolong is an assembly language [2] for the Java virtual machine, based on the Jasmin language by Jon Meyer [2]. It is designed to allow us to write programs at the Byte-code level without having to mess about with individual bytes. The Java Virtual Machine uses a binary format called class files for communicating programs. Because it is difficult to edit and visualize class files, the Oolong language was created. It is nearly equivalent to the class file format but easier to read and write. The Oolong language takes care of certain bits-and-bytes-level details while allowing almost complete control of the class file. A simple Oolong program is given in Figure 4.1.

```
1:  .class public Hello
2:  .super java/lang/Object
3:   .method public static main([Ljava/lang/String;)V
4:    .limit stack 2
5:    .limit locals 1
6:     getstatic java/lang/System/out
          Ljava/io/PrintStream;
7:     ldc "Hello, world"
8:     invokevirtual java/io/PrintStream/println
          (Ljava/lang/String;)V
9:     return
10:   .end method
11:  .end class
```

Figure 4.1 A simple Hello world Oolong Program

In the Oolong Language Lines that begin with periods (.) are called directives. The first directive is `.class`, which tells the Oolong assembler the name of the class being compiled. In the above example, the name of the class being declared is Hello. The next line after the `.class` directive contains a .super directive. The `.super` directive tells Oolong the name of the super class of this class. This above example declares the `superclass` to be `java/lang/Object`, which is the default. The `.super` directive works like extend clause in a Java program. The `.method` directive marks the beginning of a new method. Every line after that is a part of the method until the .end method

directive. The `.method` directive names the method being created [2]. In this case, the method is named main. Before the method name is a list of access keywords, which control how the class can be used. This method is marked public and static. In the Oolong language, the arguments and return types are written together in the descriptor following the method name. A method descriptor [2] contains the types of the arguments between parentheses, followed by the return type. The descriptor of *main* is `([Ljava/lang/String;)V`. This says that main takes an array of strings, and returns a void.

4.4 Microsoft Intermediate Language Code Generation

To generate Byte-code for Common Language Runtime [8] we use Microsoft Intermediate Language as an intermediate language. The MsilCodegenVisitor visit the AST [3] class hierarchy and generate the MSIL [18] code from the C- source program. There is an intermediate language assembler available with Microsoft dot net framework. The Intermediate Language assembler assembles the MSIL code to portable executable code, which is the Byte-code for the Common Language Runtime [9]. The used MSIL instructions are given in Table 4.4.

4.4.1 The Common Language Runtime

The .NET Common Language Runtime (CLR) [9] is designed to be a language-neutral architecture. The CLR differs from the JVM in this one respect. However, there are many similarities: The CLR (like the JVM). The .NET Common Language Runtime consists of a typed, stack-based intermediate

Table 4.4

Used MSIL instructions and description

Instructions	Description
.class	Class directive name of the class
.filed	Field declaration (global variable in C-)
.method	Represent method new method declaration
Call	Call a method
Ret	Return from method, possibly returning a value
Newarr	Create a zero based, one dimensional array
Brzero target	Branch to target if value (stack top) is zero
Neg	Negate the value
Clt	Compare less than
ble target	Branch to target on less than or equal to
Cgt	Compare greater than
bge target	Branch to target on greater than or equal to
Ceq	Compare equal
Bne.un target	Branch to target on not equal or unordered
Add	Add numeric values
Sub	Subtract numeric values
Mul	Multiply values
Div	Divide values
ldc.i4 n	Push n onto the stack as int32
Ldstr	Load a literal string
Stloc	Pop value from stack to local variable
Stsfld	Store a static field of a class
Ldloc	Load local variable onto the stack
Ldarg	Load argument onto the stack
Ldsfld	Load static field of a class
Stelem.i4	Replace array element at index with the int32 value on the stack
Stelem.ref	Replace array element at index with the ref value on the stack
Ldelem.i4	Load the element with type int32 at index onto the top of the stack as an int32
Ldelem.ref	Load the element with type int32 at index onto the top of the stack as an 0

language (IL), an Execution Engine (EE) [14] which executes IL and provides a variety of runtime services (storage management, debugging, profiling, security, etc.), and a set of shared libraries (.NET Frameworks). The CLR has been successfully targeted by a variety of source languages, including C#, Visual Basic, C++, Eiffel, Cobol, Standard ML [12], Mercury, Scheme and Haskell.

The primary focus of the CLR is object-oriented languages, and this is reflected in the type system, the core of which is the definition of classes in a single-inheritance hierarchy together with Java style interfaces. Also supported are a collection of primitive types, arrays of specified dimension, structs (structured data that is not boxed, *i.e.* stored in-line), and safe pointer types for implementing call-by-reference and other indirection-based tricks. Memory safety enforced by types is an important part of the security model of the CLR, and a specified subset of the type system and of IL, programs can be guaranteed type safe by verification rules that are implemented in the runtime. However, in order to support unsafe languages like C++, the instruction set has a well-defined interpretation independent of static checking, and certain types (C-style pointers) and operations (block copy) are never verifiable. IL is not intended to be interpreted; instead, a variety of native code compilation strategies is supported. Frequently used libraries such as the base class library and GUI frameworks are precompiled to native code in order to reduce start-up times. User code is typically loaded and compiled on demand by the runtime.

Common Language Infrastructure [12] of CLR uses Common Language Specifications to bind different Languages in an agreement to access frameworks by implementing at least those parts of the Common Type System (CTS) that are part of the Common Language Specifications (CLS). Hence, all the languages (C#, VB.NET, Effil.NET etc.) which are targeted towards .NET framework converse to standard single language known as Microsoft Intermediate Language (MSIL) [18].

4.4.2 Overview of Microsoft Intermediate Language

Microsoft Intermediate Language (MSIL) [18] represents the transient stage in the process of conversion of source code written in any .NET language to machine language as a pseudo-assembly language code that's between the source code you write-such as Visual Basic .NET or C#-and Intel-based

assembly language or machine code. When you compile a .NET program, , the compiler translates your source code into Microsoft intermediate language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code. When we execute the code, MSIL is converted to CPU-specific code, usually by a just-in-time (JIT) compiler. Because the common language runtime supplies one or more JIT compilers, the same set of MSIL can be JIT-compiled and executed on any supported architecture.

It is obviously imperative to gain mastery over Intermediate Language, because knowledge of Intermediate Language translates into competence over IL code that may have originally been written in any programming language. MSIL (or simply IL) puts an end to the unending war amongst programmer community on the superiority of one language over others. To this end, IL is a great leveler. In .NET, world one part of the code may be written in Effil while other may have been written in C# or VB.NET, but it all eventually gets converted in IL. This provides great freedom and flexibility to the programmers to select the language she is more familiar with and does away with the need to constantly learning new languages every day. A simple HelloWorld program written in Microsoft Intermediate language is given in Figure 4.2.

```
1:    .assembly extern mscorlib
2:    {
3:      .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
4:      .ver 1:0:5000:0
5:    }
6:   .assembly HelloWorld
7:    {
8:      .hash algorithm 0x00008004
9:      .ver 0:0:0:0
10:   }
11: .class private auto ansi beforefieldinit HelloWorld
      extends [mscorlib]System.Object
12:   {}

13: .class private auto ansi beforefieldinit HelloWorld
      extends [mscorlib]System.Object
```

```
14:  {
15:   .method private hidebysig static void  Main(string[]
                                        args)cil managed
16:    {
17:     .entrypoint
18:     .maxstack  1
19:      ldstr   "Hello World"
20:      call void
              [mscorlib]System.Console::WriteLine(string)
21:      ret
22:    }
23:  }
```

Figure 4.2: A simple HelloWorld program written in MSIL

In the Microsoft Intermediate Language anything that begins with dot "." Is a directive to the assembler, asking it to perform some functioning, such as creating a function or class. Whereas anything that does not start with a dot "." is an actual instruction. The first directive is `.assembly extern mscorlib` and the second directive `assembly HelloWorld` is represents the assembly manifest. The MSIL code is always compiles to one module and a module always associate with one assembly. The concept of modules and assembly are extremely crucial in .NET world and should be thoroughly understood. The next directive is the `class structure` declaration and this class is inheriting the `System.Object` class. The `private` is accessibility attribute. The `auto` attribute means that the layout of the class in memory will be decided by the runtime and not by the program. The attribute `ansi` is used for smooth transition between unmanaged and managed code. Code, which is not targeted towards Common Language Infrastructure, is termed as unmanaged code. Languages like C, C++ or VB6 all produce unmanaged codes. We need an attribute that handle the interoperability between managed and unmanaged codes. In managed code, a string is represented as 2-byte Unicode characters, where as unmanaged code uses 1-byte ANSI characters, attribute `ansi` is used to handle transition of strings from one code to another. The next directive represents the `class definition`. In the class definition, the first directive is for method

definition. A function in MSIL is created by assembler directive `method` it followed by return type of method, in our case void and then actual name of the function, HelloWorld with the pair of round brackets "`()`". The start point and the end point of the functions are signified by the curly brackets "`{}`". A well-formed function written in IL always has the "`end of function`" instruction that is "`ret`". The attribute `hidebysig` ensures that the function in the parent class is hidden from the derived class having same name and signature. In our example, this attribute make sure that if the function HelloWorld is present in the base class, it is not visible in derived class. In MSIL, the first function to be start executed from function `entrypoint`.

4.5 Code Improvement

Code improvement is often referred to as optimization, though it seldom makes anything optimal in any absolute sense. It is an optional phase of compilation whose goal is to transform a program into a new version that computes the same result more efficiently, more quickly or using less memory or both [19].

Some improvements are machine independent. These can be performed as transformation on the intermediate form such as AST or three-address code. Other improvements require an understanding of target machines. These must be performed as transformation on the target program. Thus code improvement often appears as two additional phases of compilation, one immediately after semantic analysis and intermediate code generation, the other immediately after target code generation [5].

We yet not perform any code improvement. This will be our future implementation. Now, we will discuss on the form of code improvement that tend to achieve the largest increase in execution speed, and are most widely used. Most of available compiler generate control flow graph (CFG) form intermediate

representation like AST or generate another low-level intermediate language such as three-address code then CFG.

We can create a CFG from AST in which each node contains a linear sequence of three-address instruction for and idealized machine, typically one with one with unlimited supply of virtual register. The machine specific part of back-end begins with target code generation. This phase string the basic block together into a liner program, translating each block into the instruction set of the target machine and generating branch instructions that correspond to the arcs of the control flow graph. We may also transform our AST to an intermediate language called Control-Oriented Register transfer language (CORTL) [18] or CORTL AST instead of CFG. However, Generating CORTL from AST is costly.

In the machine-independent code improvement, we can eliminate redundant loads, store, and computation within each basic block. The second deals with similar redundancies across the boundaries between basic block. The third effects several improvements specific to loops; these are particularly important, since most program spend most of their time in loops.

Chapter V

CONCLUSION

In this chapter, we will summarize the foregoing chapters and evaluate the result of our work. Finally, we will conclude by discussing some possibilities for future related work.

5.1 Summary

In this thesis, we have described the C- compiler and its working procedure. The C- compiler supports the C- language. The C- language is like C but with out pointer, structure and some control iteration statements. The C- language supports only integer, void and string. This language does not allow any string operations. There are some other differences also available between the C and C-.

To implement the C- compiler we used the Java programming language. Our compiler takes a C- source code and performs the lexical analysis by the Jlex, which is a lexical analyzer that supports Java language. During syntax analysis, we store the tokens information. The lexical analyzer provide valid token to the syntax analyzer that check the syntax of C- source code. We use JCUP as a syntax analyzer. The syntax analyzer takes token as input, analyze by the C- language grammar, and build the parse tree. During syntax analysis, we contract the Abstract Syntax tree instead of parse tree. Our AST is a Java class hierarchy that holds the necessary information of C- source program.

To analyze the semantics of C- program we have used the visitor pattern. We split the semantics analysis into name check and type check. To name and type check we have used NameVisitor and TypeVisitor. Each visitor visits the class hierarchy and performs the operations. To name check we use symbol table that holds information about the symbols. Our symbol table is a collection of hash table into a list. The symbol information contains the symbol name, type, and the scope of the symbol. The type check visitor visits the hierarchy and check the type of variables, statements, and expressions

The final phase of our compiler model is the code generation. Our C- compiler generates codes for multi target machine. Our target machines are SPIM simulator, Java Virtual Machine, and Common Language Runtime. We generate MIPS assembly language for SPIM simulator, Oolong assembly language for Java virtual Machine, and Microsoft Intermediate Language for Common Language Runtime. Oolong and MSIL are the intermediate code generation. To generate Java byte- code from Oolong code we have to assemble by the Oolong assembler, similarly to generate CLR Byte-code we have to assemble the MSIL by the Microsoft intermediate language assembler.

5.2 Future Work

In this section, we will discuss potential direction for future work on this compiler.

5.2.1 Code optimization

For the time limitation we cannot optimized our generated code. We will do redundancy elimination in basic blocks, peephole optimization, global common sub-expression elimination, and some loop improvement .Moreover we will concentrate on the form of code improvement that tend to achieve the largest increase in execution speed, and are most widely use.

5.2.2 Extension of the compiler

Since our C-, language supports a subset of C language; a substantial amount of work must still be done to expand our compiler to cover the entire C language. We have mentioned some of the unimplemented feature in our discussion of C- language grammar especially all the data type, pointer and records. We hope that in future we will implement a compiler that will generate codes for multi target machine from the entire C language.

BIBLIOGRAPHY

Books:

[1] Kenneth C.Louden, Compiler Construction Principles and Practice, PWS Publishing Company, 1997.

[2] Joshua Engel, Programming for Java™ Virtual Machine, Addison Wesley, 1999.

[3] Frank Yellin, Tim Lindholm , The Java™ Virtual Machine Specification, Sun Microsystems, Inc., 1999.

[4] Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman, Compilers Principles, Techniques, and tools, Pearson Education Asia, 2004/2005.

[5] Michael L. Scott, Programming Language Pragmatics, Harcourt Asia PTE Ltd., Morgan Kaufmann, 2000.

[6] Steven S. Muchnick, Advanced Compiler Design Implementation, Morgan KaufMann, 2003.

[7] John L.Hennessy, David A. Pattterson, Computer Organization and Design, Hardcourt (India) Private Limited & Morgan Kaufmann,1999.

Papers:

[8] Eric Meijer, John Gough, Technical Overview of the Common Language Runtime. MIT, 2001.

[9] Andrew Kennedy, Don Syme, Design and Implementation of Generic for the .NET Common Language Runtime, Microsoft Corporation, 2000.

[10] Jermy Singer, JVM versus CLR: A Comparative Study. University of Cambridge. 2001.

[11] Jermy Singer Carl D. Mcconnell, Tree Based Code Optimization, University of Illinois at Urbana-Champaign, 1993.

Drafts:

[12] ECMA TC39/TG3, Common Language Infrastructure Partition I: Concepts and Architecture, 2002.

[13] ECMA TC39/TG3, Common Language Infrastructure Partition III: CIL Instruction Set, 2002.

Websites:

[14] Jlex: A lexical analyzer generator for Java ™ documentation. www.cs.princeton.edu/~appel/modern/java/JLex

[15] Jcup: A parser generator for Java™ documentation. www.cs.princeton.edu/~appel/modern/java/CUP

[16] Jasmin: A Java assembler for the Java Virtual machine documentation. http://jasmin.sourceforge.net/

[17] SPIM: A MIPS 32 Simulator documentation. http://www.cs.wisc.edu/~larus/spim.html

[18] Demystifying Microsoft Intermediate Language documentation. http://www.devcity.net/Articles/54/msil_1_intro.aspx

[19] Lcc: A Retargetable Compiler for ANSI C http://www.cs.princeton.edu/software/lcc/

[20] Axiomatic Multi-platform C http://www.axiomsol.com/

[21 www.javaworld.com/javaworld/javatips/jw-javatip98.html

APPENDIX A

Code of Lexical Analysis

In this appendix, we will add the source code of lexical analysis. The complete source code can be found on the web page at http://

```
/* cminus.jlex
 * Author: Md.Zahurul Islam <zaisbd@yahoo.com>
 * (C) Copyright 2005 Zahurul Islam

 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.

 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */
```

```
/*
 *Java LEX specification for a parser for C-- programs
 */

import java_cup.runtime.Symbol;

/**  The generated scanner will return a Symbol for each token that it finds.
 *   A Symbol contains an Object field named value; that field will be of type
 *   TokenInfo, defined below.
 *
 *   A TokenInfo object contains the line number on which the token occurs as
 *   well as the number of the character on that line that starts the token.
 *   Some tokens (e.g., literals) also include the value of the token.
 */

class StrDequote {
    private int pos1,pos2;
    private String result;
   String dequote(String s,int l,int c) {
    pos1 = s.indexOf("\"");
    pos2 = s.lastIndexOf("\"");
    if (pos1 == -1 || pos2 == -1)
      Report.warning(l,c,"String literal \""+ s +"\" not correctly quoted");
    result = s.substring(pos1+1,pos2);
    return result;
 }
}

class TokenInfo {
   // fields
   int linenum;
   int colunnum;
   String lexem;
   // constructor
   TokenInfo(int l, int c, String lex) {
     linenum = l;
     colunnum = c;
     lexem = lex;
   }
}

//TokenInfo for Integer
class IntLitTokenInfo extends TokenInfo {
   // new field: the value of the integer literal
   int intVal;
   // constructor
   IntLitTokenInfo(int l, int c, String str) {
     super(l, c,str);
```

```
    try {
      intVal = (new Integer(str)).intValue();
    } catch (NumberFormatException e) {
      Report.warning(linenum,colunnum, "integer literal too large; using max value");
      intVal = Integer.MAX_VALUE;
    }
  }
}

//TokenInfo for both STRINGLITERAL and ID
class StringTokenInfo extends TokenInfo {
  String strVal;

  StringTokenInfo(int l, int c, String s) {
    super(l, c,s);
  strVal = s;
  }

}

%%
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%char
%line

%eofval{
return new Symbol(sym.EOF);
%eofval}

ws              =       [ ]
tabs        =       [\t]
newline                 =       [\n]
letter          =       [A-Za-z]
digit           =       [0-9]
identifier      =       {letter}({letter}|{digit}|_)*
integer         =       {digit}+
string      =       \"[^\"]*\"
comment                 =       "//"
%%

"int"           { return new Symbol(sym.T_INT, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"bool"          { return new Symbol(sym.T_BOOL, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"string"{ return new Symbol(sym.T_STRING, new TokenInfo(yyline + 1,
```

```
            yychar,yytext()));
          }
"void"        { return new Symbol(sym.T_VOID, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"true"        { return new Symbol(sym.T_TRUE, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"false"       { return new Symbol(sym.T_FALSE, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"if"        { return new Symbol(sym.T_IF, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"else"        { return new Symbol(sym.T_ELSE, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"while"       { return new Symbol(sym.T_WHILE, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"return"      { return new Symbol(sym.T_RETURN, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }

"+"           { return new Symbol(sym.T_PLUS, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"-"           { return new Symbol(sym.T_MINUS, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"*"           { return new Symbol(sym.T_MULT, new TokenInfo(yyline + 1,
           yychar,yytext()));
          }
"/"           { return new Symbol(sym.T_DIV, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"<"           { return new Symbol(sym.T_LT, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"<="          { return new Symbol(sym.T_LE, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
">"           { return new Symbol(sym.T_GT, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
">="          { return new Symbol(sym.T_GE, new TokenInfo(yyline + 1,
            yychar,yytext()));
          }
"=="          { return new Symbol(sym.T_EQ, new TokenInfo(yyline + 1,
```

```
                yychar,yytext()));
            }
"!="        { return new Symbol(sym.T_NE, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
"&&"        { return new Symbol(sym.T_AND, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
"||"        { return new Symbol(sym.T_OR, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
"!"         { return new Symbol(sym.T_NOT, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
"="         { return new Symbol(sym.T_ASSIGN, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
";"         { return new Symbol(sym.T_SEMI, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
","         { return new Symbol(sym.T_COMMA, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
"("         { return new Symbol(sym.T_LPAREN, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
")"         { return new Symbol(sym.T_RPAREN, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
"["         { return new Symbol(sym.T_LBRACK, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
"]"         { return new Symbol(sym.T_RBRACK, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
"{"         { return new Symbol(sym.T_LBRACE, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }
"}"         { return new Symbol(sym.T_RBRACE, new TokenInfo(yyline + 1,
                yychar,yytext()));
            }


{identifier}    { return new Symbol(sym.T_ID, new StringTokenInfo(yyline+1,
                yychar,yytext()));
            }
{integer}       { return new Symbol(sym.T_INT_LITERAL,new IntLitTokenInfo(yyline+1,
                yychar, yytext()));
            }
```

```
{string}{
              StrDequote str = new StrDequote();
              String dstr = str.dequote(yytext(),yyline+1,yychar);
              return new Symbol(sym.T_STR_LITERAL,new StringTokenInfo(yyline+1,
              yychar, dstr));
            }

{newline}+      { yychar = 0; }

{ws}+          { }

{tabs}+        { }

{comment}[^\n]* { }
.              {  Report.error(yyline+1, yychar, "Illegal token ::" + yytext());

              }
```

Appendix B

Code of Syntax Analyzer

```
/* cminus.jlex
 * Author: Md.Zahurul Islam <zaisbd@yahoo.com>
 * (C) Copyright 2005 Zahurul Islam

 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.

 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

/*
 *Java LEX specification for a parser for C-- programs
 */

import java_cup.runtime.Symbol;
```

```
/**  The generated scanner will return a Symbol for each token that it finds.
 *   A Symbol contains an Object field named value; that field will be of type
 *   TokenInfo, defined below.
 *
 *   A TokenInfo object contains the line number on which the token occurs as
 *   well as the number of the character on that line that starts the token.
 *   Some tokens (e.g., literals) also include the value of the token.
 */

class StrDequote {
    private int pos1,pos2;
    private String result;
   String dequote(String s,int l,int c) {
    pos1 = s.indexOf("\"");
    pos2 = s.lastIndexOf("\"");
    if (pos1 == -1 || pos2 == -1)
      Report.warning(l,c,"String literal \""+ s +"\" not correctly quoted");
    result = s.substring(pos1+1,pos2);
    return result;
 }
}

class TokenInfo {
   // fields
   int linenum;
   int colunnum;
   String lexem;
   // constructor
   TokenInfo(int l, int c, String lex) {
     linenum = l;
     colunnum = c;
     lexem = lex;
   }
}

//TokenInfo for Integer
class IntLitTokenInfo extends TokenInfo {
   // new field: the value of the integer literal
   int intVal;
   // constructor
   IntLitTokenInfo(int l, int c, String str) {
     super(l, c,str);
     try {
       intVal = (new Integer(str)).intValue();
     } catch (NumberFormatException e) {
       Report.warning(linenum,colunnum, "integer literal too large; using max value");
       intVal = Integer.MAX_VALUE;
     }
  }
```

```
}

//TokenInfo for both STRINGLITERAL and ID
class StringTokenInfo extends TokenInfo {
  String strVal;

  StringTokenInfo(int l, int c, String s) {
    super(l, c,s);
  strVal = s;
  }

}

%%
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%char
%line

%eofval{
return new Symbol(sym.EOF);
%eofval}

ws              =       [ ]
tabs        =       [\t]
newline                 =       [\n]
letter          =       [A-Za-z]
digit           =       [0-9]
identifier      =       {letter}({letter}|{digit}|_)*
integer         =       {digit}+
string      =       \"[^\"]*\"
comment                 =       "//"
%%

"int"           { return new Symbol(sym.T_INT, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"bool"          { return new Symbol(sym.T_BOOL, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"string"{ return new Symbol(sym.T_STRING, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"void"          { return new Symbol(sym.T_VOID, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"true"          { return new Symbol(sym.T_TRUE, new TokenInfo(yyline + 1,
              yychar,yytext()));
```

```
            }
"false"         { return new Symbol(sym.T_FALSE, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"if"        { return new Symbol(sym.T_IF, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"else"          { return new Symbol(sym.T_ELSE, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"while"         { return new Symbol(sym.T_WHILE, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"return"        { return new Symbol(sym.T_RETURN, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }

"+"             { return new Symbol(sym.T_PLUS, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"-"             { return new Symbol(sym.T_MINUS, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"*"             { return new Symbol(sym.T_MULT, new TokenInfo(yyline + 1,
             yychar,yytext()));
            }
"/"             { return new Symbol(sym.T_DIV, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"<"             { return new Symbol(sym.T_LT, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"<="            { return new Symbol(sym.T_LE, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
">"             { return new Symbol(sym.T_GT, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
">="            { return new Symbol(sym.T_GE, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"=="            { return new Symbol(sym.T_EQ, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"!="            { return new Symbol(sym.T_NE, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"&&"            { return new Symbol(sym.T_AND, new TokenInfo(yyline + 1,
              yychar,yytext()));
```

```
            }
"||"            { return new Symbol(sym.T_OR, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"!"             { return new Symbol(sym.T_NOT, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"="             { return new Symbol(sym.T_ASSIGN, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
";"             { return new Symbol(sym.T_SEMI, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
","             { return new Symbol(sym.T_COMMA, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"("             { return new Symbol(sym.T_LPAREN, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
")"             { return new Symbol(sym.T_RPAREN, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"["             { return new Symbol(sym.T_LBRACK, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"]"             { return new Symbol(sym.T_RBRACK, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"{"             { return new Symbol(sym.T_LBRACE, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }
"}"             { return new Symbol(sym.T_RBRACE, new TokenInfo(yyline + 1,
              yychar,yytext()));
            }


{identifier}    { return new Symbol(sym.T_ID, new StringTokenInfo(yyline+1,
              yychar,yytext()));
            }
{integer}       { return new Symbol(sym.T_INT_LITERAL,new IntLitTokenInfo(yyline+1,
              yychar, yytext()));
            }

{string}{
              StrDequote str = new StrDequote();
              String dstr = str.dequote(yytext(),yyline+1,yychar);
              return new Symbol(sym.T_STR_LITERAL,new StringTokenInfo(yyline+1,
              yychar, dstr));
            }
```

```
{newline}+      { yychar = 0; }

{ws}+           { }

{tabs}+         { }

{comment}[^\n]* { }
.               {  Report.error(yyline+1, yychar, "Illegal token ::" + yytext());

                }
```