SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

INTERMEDIATE CODE GENERATION AND OPTIMIZATION

Gudied by:

Dr.K.Vijaya

Submitted By:

Adithya Anup(RA2011003010025)

Yukendhira.M(RA2011003010052)

Aim:-

To implement the front end of a compiler that generates three address code (Quadruple,Triple,Indirect triple).

ABSTRACT:-

A compiler is a software program that converts source code written in one programming language to another programming language. A compiler typically consists of two parts: the front end and the back end. The front end is responsible for analyzing the source code, checking its syntax, and generating an intermediate representation of the program, such as three address code. The back end is responsible for optimizing the intermediate representation and generating machine code for a specific target architecture.

In this project, we will be implementing the front end of a compiler using C++ code. Our compiler will take as input a program written in a simple programming language and generate three address code in three different formats: quadruple, triple, and indirect triple.

Front End Implementation:

The front end of our compiler will consist of three main components: a lexer, a parser, and a code generator.

Lexer:

The lexer is responsible for breaking down the source code into a sequence of tokens, such as keywords, identifiers, operators, and constants. We will be using regular expressions to match the patterns of the tokens.

Code Generator:

The code generator is responsible for generating three address code from the AST. We will be using three different formats for the three address code: quadruple, triple, and indirect triple.

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

Target language : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.

IR Type : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.

Selection of instruction : The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.

Register allocation : A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

Ordering of instructions : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:

Register descriptor : Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.

Address descriptor : Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Programming Language:

The programming language we will be using for this project is a simple language that supports the following features:

Arithmetic expressions with operators +, -, *, /

Assignment statements with variable names and constant values

Conditional statements with if-else constructs

Looping constructs with for and while loops

Intermediate Code Generation:

Intermediate code generation is the process of transforming the source code into an intermediate representation that is easier to analyze and optimize. This intermediate representation is typically a lower-level language that is closer to machine code, but still independent of the target machine's architecture. The intermediate code can be optimized and transformed into machine code during the back-end phase of the compiler.

Types of Intermediate Code:

There are several types of intermediate code, including:

Three Address Code:

Three Address Code (TAC) is a low-level intermediate code that uses at most three operands per instruction. It is easy to generate and manipulate, making it a popular choice for compilers.

Quadruples:

Quadruples are similar to TAC, but each instruction has four operands. The additional operand is typically used to hold the result of the operation.

Triples:

Triples are a more compact representation of TAC. Each instruction has three operands, and the result is stored in the left operand.

Intermediate Code Generation Process:

The intermediate code generation process typically follows these steps:

Lexical Analysis:

The source code is broken down into a sequence of tokens, which are then used to build a parse tree.

Syntax Analysis:

The parse tree is used to build an Abstract Syntax Tree (AST), which represents the structure of the program.

Semantic Analysis:

The AST is analyzed to ensure that it conforms to the language's syntax and semantics. This phase also includes type checking and error detection.

Intermediate Code Generation:

The AST is traversed to generate intermediate code. The intermediate code can be in the form of TAC, quadruples, or triples.

Optimization:

The intermediate code is optimized to improve its performance and reduce its size. This phase can include dead code elimination, constant folding, and loop optimization.

Code Generation:

The optimized intermediate code is transformed into machine code for the target machine's architecture.

.Advantages of Intermediate Code

- We need intermediate code because if we don’t have the option to create intermediate code, we require a native compiler for each new machine.
- It becomes very easy to apply source code changes to enhance the implementation by optimizing the intermediate code.
- Intermediate code supports eliminating the requirement of a new complete compiler for every individual machine by upholding the same analysis part for all the compilers.
- It becomes more effortless to use the source code transformations to enhance code implementation by using code optimization strategies on the intermediate code.

Benefits of Intermediate Code Generation:

- Consider that a compiler translates a source program directly into a target program. And it doesn’t generate an intermediate code between. Here we can execute the generated target code only on the machine on which the compiler was executed.
- In this case, we need a native compiler for each of the new machines.
- The intermediate code wipes out the need for the full native compiler for each machine.

The intermediate code generation phase plays a critical role in improving the efficiency and performance of the final executable code. It allows for the optimization of the code and reduces the complexity of the compilation process. Additionally, intermediate code generation also facilitates the implementation of language features such as exception handling, garbage collection, and dynamic linking.

Three-Address Code

The three-address code is also a form of intermediate representation like a syntax tree. The parser generates a syntax tree. And the intermediate code generator generates the three-address code. This intermediate representation is a sequence of assembly-like instructions.
Note: It is possible for the compiler to construct a syntax tree and three-address code parallely.

Three-Address Instructions

Three-address instruction has three operands per instruction. And each operand act like a register.

1. The three-address assignment instruction must have at least one operand on the right side of the instruction.
2. To hold the value computed by the three-address code instruction, the compiler generates a temporary name.
3. The three-address instructions may have less than three operands.

The three-address instruction is of the form:
x = y **op** z
Here,
y and z are the operands.

op is the operator.

x can be a name or a compiler-generated temporary name (location) that will hold the result.

Types of Intermediate Code

We can classify the intermediate representation into two types:

1. **High-Level Intermediate Representation**

This representation is the very first intermediate representation of the source language. The syntax tree generated by the parser is the high-level representation.

The high-level intermediate representation provides a hierarchical structure of the source program. This hierarchical structure helps the task such as a static check

.

2. **Low-Level Intermediate Representation**

After high-level intermediate representation, the next intermediate representation is low-level intermediate representation. The intermediate code generation phase generates the low-level intermediate representation.

The three-address code is a low-level representation. The three-address code is suitable for machine-dependent tasks. Like register allocation and instruction selection.

Three-Address Instructions

The three-address instructions are always executed sequentially. They can even perform the conditional or unconditional jump.

Address and Instruction

The three-address code is based on two concepts addresses and the instruction.

Address:

1.Name: A name specifies the identity of a variable in the instructions or even the name of the source program. In the three-address code, the name in the instruction appears as the address.

2. Constant: There can be different types of constants that the compiler deal with. The compiler performs the necessary type of conversion when required.

3. Compiler-generated temporary: t is a name generated by the compiler. The compiler generates it each time there is a need for a temporary variable. This address stores the result or a value.

Instructions:

1. **Assignment Instructions**

- **x = y op z**, here the x, y, and z are addresses and op can be an arithmetic or logical binary operator.|

- **x = op y**, where x and y are the addresses and op is a unary operator.

2. **Copy Instruction**
The copy instruction **x = y**, where x and y are addresses and the value at the address of y is set at the address of x.

3. **Jump Instruction**

The conditional jump, **if x goto L** and **ifFalse x goto K**. Here the instruction with label L is executed if x is true. And if x is false the instruction with label K is executed.

The conditional jump if x **relop** y goto L, here **relop** is the relational operator. And if x stands by the relational operator then the instruction with label L is executed. If not, the instruction next to this conditional instruction is operated.

Unconditional jump goto L, here the instruction with label L is executed next.

Advanced Instructions:

1. **Procedure Call**
The instruction of the form **y = call p**, n make a procedure or function call. Here p is the name of the procedure and n is the integer that indicates the number of parameters passed. The instruction of the form '**return y**' here y indicates the returned value

.

2. **Indexed Copy Instruction**
The instruction of form **x = y[i]**, where the value at the location ith memory unit beyond the location y is set in x. In the instruction **x[i] = y**, the value of y is set to the ith memory location beyond the location of x.

3. **Address and Pointer Assignment**
In the instruction of form **x = &y**, here the L value of y is set to x. The instruction **x = *y**, here the r-value of y holds L value of a location say z. And the r-value of z is set to x. The instruction ***x = y** sets the r-value of y to the location pointed by x.

The intermediate representation must be:

1. Easy to produce.
2. Easy to translate into target code.

Strength reduction Optimization

- Strength reduction optimization is a technique used in compiler optimization to improve the performance of code by replacing expensive operations with cheaper ones.

- The goal of this optimization is to reduce the number of operations needed to perform a given task, thus improving the overall efficiency of the code.

Code Before Optimization	Code After Optimization
B = A x 2	B = A+A

Requirements to run the script:

Hardware: A computer or device with a compatible operating system (e.g., Windows, macOS, Linux) and at least 2 GB of RAM.

Software: CLion 2022.3.2 or later installed on the system.

Input data: The script may require input data in a specific format or located in a particular directory. Ensure that the input data is available and properly formatted before running the script.

Output location: The script may generate output files that need to be saved in a specific location or with a specific name. Make sure that the output directory exists and has write permissions before running the script.

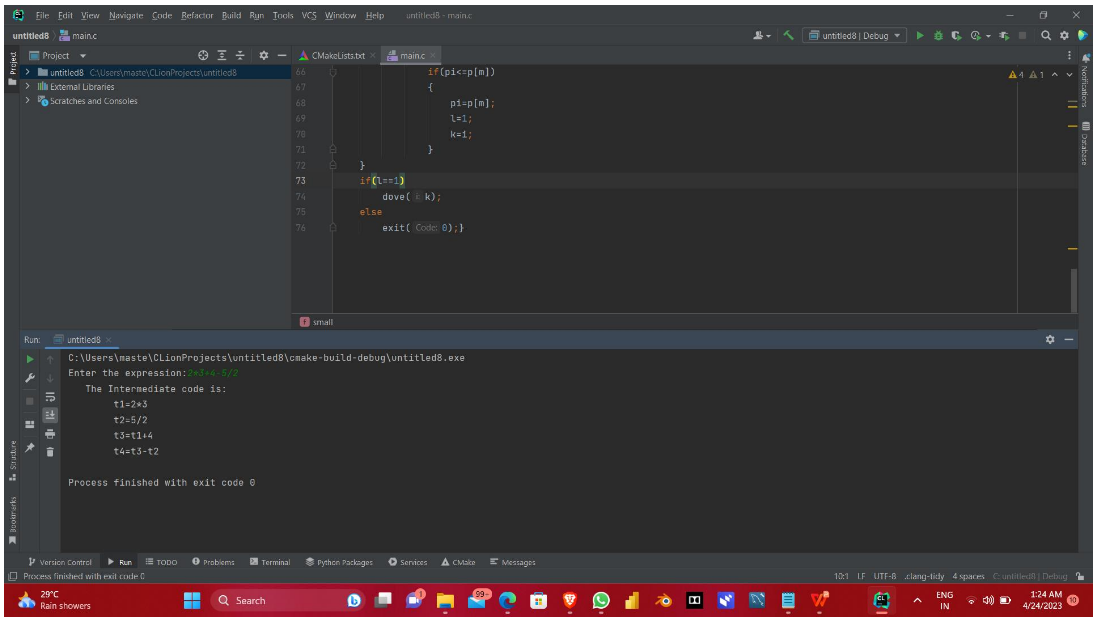
Code:

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

void scan();
void shift(int i);

int p[5] = {0, 1, 2, 3}, c = 1, i, k, l, m, pi;
char sw[5] = { '-', '+', '/', '*'}, j[20], a[5], b[5], ch[2];

void main()
{
   printf("Enter the expression:");
   scanf("%s", j);
   printf("\tThe Intermediate code is:\n");
   scan();
}

void shift(int i)
{
   a[0]=b[0]='\0';
   if(!isdigit(j[i+2])&&!isdigit(j[i-2]))
   {
    a[0]=j[i-1];
    b[0]=j[i+1];
}
if(isdigit(j[i+2])){
    a[0]=j[i-1];
    b[0]='t';
    b[1]=j[i+2];
}
if(isdigit(j[i-2]))
{
    b[0]=j[i+1];
    a[0]='t';
    a[1]=j[i-2];
    b[1]='\0';
}
if(isdigit(j[i+2]) && isdigit(j[i-2]))
{
    a[0]='t';
    b[0]='t';
    a[1]=j[i-2];
    b[1]=j[i+2];
    sprintf(ch,"%d",c);
    j[i+2]=j[i-2]=ch[0];
}
if(j[i]=='*') {
    // check if the expression is multiplying by 2
    if(b[0] == '2' && b[1] == '\0') {
        // replace multiplication with shift left by 1
        printf("\tt%d=%s<<1\n",c,a);
    } else {
        // use normal multiplication operator
        printf("\tt%d=%s*%s\n",c,a,b);
    }
}
if(j[i]=='/') {
    // check if the expression is dividing by 2
    if(b[0] == '2' && b[1] == '\0') {
        // replace division with shift right by 1
        printf("\tt%d=%s>>1\n",c,a);
    } else {
        // use normal division operator
        printf("\tt%d=%s/%s\n",c,a,b);
    }
}
if(j[i]=='+') {
    // add operation
    printf("\tt%d=%s+%s\n",c,a,b);
}
if(j[i]=='-') {
    // subtract operation
    printf("\tt%d=%s-%s\n",c,a,b);
}
```

```
    if(j[i]=='=') {
        // assignment operation
        printf("\t%c=%s\n",j[i-1],a);
    }
    sprintf(ch,"%d",c);
    j[i]=ch[0];
    c++;
    scan();
}
void scan()
{   pi=0;l=0;
    for(i=0;i<strlen(j);i++)
    {       for(m=0;m<5;m++)
            if(j[i]==sw[m])
                if(pi<=p[m])
                {
                    pi=p[m];
                    l=1;
                    k=i;
                }
    }
    if(l==1)
        shift(k);
    else
        exit(0);
}
```

Output:

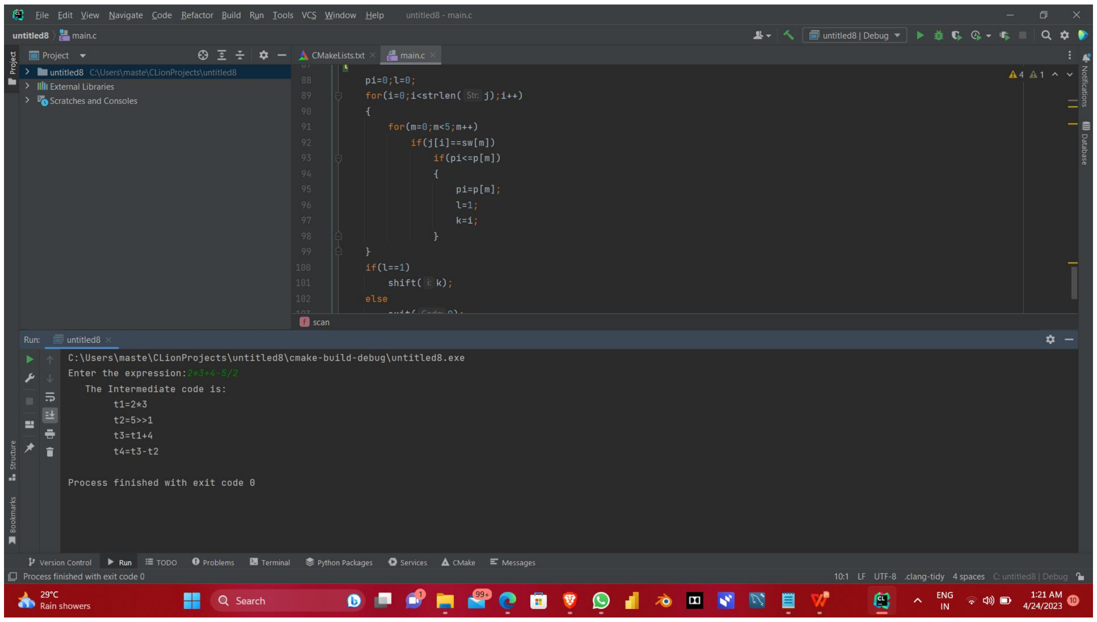
Code Before optimization:



```
                if(pi<=p[m])
                {
                    pi=p[m];
                    l=1;
                    k=i;
                }
    }
    if(l==1)
        dove( k);
    else
        exit( 0);}
```

```
C:\Users\maste\CLionProjects\untitled8\cmake-build-debug\untitled8.exe
Enter the expression:2*3+4-5/2
   The Intermediate code is:
        t1=2*3
        t2=5/2
        t3=t1+4
        t4=t3-t2

Process finished with exit code 0
```

Code After Optimization:



```
    pi=0;l=0;
    for(i=0;i<strlen( j);i++)
    {
        for(m=0;m<5;m++)
            if(j[i]==sw[m])
                if(pi<=p[m])
                {
                    pi=p[m];
                    l=1;
                    k=i;
                }
    }
    if(l==1)
        shift( k);
    else
```

```
C:\Users\maste\CLionProjects\untitled8\cmake-build-debug\untitled8.exe
Enter the expression:2*3+4-5/2
   The Intermediate code is:
        t1=2*3
        t2=5>>1
        t3=t1+4
        t4=t3-t2

Process finished with exit code 0
```

Result:

Intermediate code generation along with strength reduction optimization is implemented successfully.