

## CS4100/5100- Introduction to Quad Codes

### SPRING 2022

In the code generation phase of the compiler project, the parser will generate Quad Codes using the data structures and machine-language opcodes discussed below. In addition, the Quad Interpreter described here will be implemented in the Java programming language. The implementation details for the required structures are outlined in the actual assignment.

#### The Quad Codes Table (an abstraction of a Code Memory Block)

Quad Codes derive their name from the fact that each one consists of four elements: an **opcode**, followed by three operands, **operand1**, **operand2** and **operand3**. By default, operand3 is typically the location of the result of the operation. The goal in this particular project is for an array of these Quads, referred to as the Quad Table, to emulate a block of real-world machine memory, and produce a useful representation of the way a real, executable machine-code file would function. The opcodes represent a very basic, though functional, machine-code language which can be easily generated from a parser, and easily interpreted for execution.

#### The Symbol Table (an abstraction of a Data Memory Block)

As with many real memory management schemes involving machine code, the Code Memory Block (i.e., the Quad Table, where instructions are stored) is distinct from the Data Memory Block (i.e., the storage area where data to be manipulated is kept). For this project, the Data Block will be implemented through the compiler's Symbol Table. This diverges from the 'real world' Data Block, in that this table is storing additional compilation data which is not necessary for the actual code execution. This is a sacrifice of literal accuracy of the model in the interest of simplifying the required programming tasks.

For ease of comprehension, at this point the Symbol Table can be thought of simply as an array of variable names and their associated values. As the compiler project progresses, the full usage of the Symbol Table during compilation and after will be expanded. By carefully building all the needed data structures according to the good software engineering practices provided in the assignments, making future extensions will be trivial. (For the curious, some of the compilation data stored in the Symbol Table will include variable *name* strings, an *element type* indicator (specifying the entry as a label, identifier, or constant, a *data type* indicator (integer, float, or string), and a *value* slot to hold the current value of the entry. For **variable identifiers**, this will be the value currently assigned to the variable; for **labels**, this will be the Program Counter location associated with this label's position in the source file; for **constants**, this will be the value corresponding to the text of the constant read from the source.

## The Op Codes

The following opcodes are defined for this Quad machine language. Most of them are related to branch instructions, and provided to simplify the code generation process. They are:

<b>TERMINATE</b>		
0	STOP	Terminate program
<b>MATH</b>		
1	DIV	Compute op1 / op2, place result into op3 (assume integer divide now)
2	MUL	Compute op1 * op2, place result into op3
3	SUB	Compute op1 - op2, place result into op3
4	ADD	Compute op1 + op2, place result into op3
<b>DATA STORAGE</b>		
5	MOV	Assign the value in op1 into op3 (op2 is ignored here)
<b>UTILITY</b>		
6	PRINT	Write symbol table name and value of op3 to StandardOutput (console)
7	READ	Read the next integer from StandardInput (keyboard) as value of op3
<b>BRANCH INSTRUCTIONS</b>		
8	JMP	Branch (unconditional); set program counter to op3 (QuadTable index, not S.T.)
9	JZ	Branch Zero; if op1 = 0, set program counter to op3
10	JP	Branch Positive; if op1 > 0, set program counter to op3
11	JN	Branch Negative; if op1 < 0, set program counter to op3
12	JNZ	Branch Not Zero; if op1 <> 0, set program counter to op3
13	JNP	Branch Not Positive; if op1 <= 0, set program counter to op3
14	JNN	Branch Not Negative; if op1 >= 0, set program counter to op3
15	JINDR	Branch (unconditional); set program counter to op3 Symbol Table value contents ( <b>indirect</b> )

Note that the use of the six different branches allows for simplified code generation later when doing mathematical comparisons. The first thing to understand is that operands are *almost always* interpreted as **indexes into the symbol table's value storage area**, which is implemented as described above. So, for example, when the following MUL Quad is executed:

MUL 2, 5, 7

it is interpreted as

SymbolTable(7).Value = SymbolTable(2).Value \* SymbolTable(5).Value

where SymbolTable is accessible as an indexed structure with a value field reference. In essence, this represents the memory area of the 'virtual machine' which interprets the quads.

Similarly,

```
MOV 3,    0,    4
```

means

```
SymbolTable(4).Value = SymbolTable(3).Value
```

Branch/jump instructions 8-14 are somewhat different, in that they **do not treat op3 as a symbol table index**, but instead treat it as an immediate value index into the QuadTable. For example,

```
JNZ 4,    0,    25
```

is interpreted in pseudocode as

```
IF SymbolTable[4].Value <> 0 THEN  
    PROGRAM_COUNTER = 25
```

The only branch instruction which is an exception to this is the JINDR (15), where

```
JINDR    0,    0,    15
```

is interpreted as

```
PROGRAM_COUNTER = SymbolTable(15).Value
```

PRINT accesses the symbol table *name* field in order to print out something intelligible, as well as the *value* of that entry. Thus

```
PRINT    8,    0,    0
```

would do this (in a higher level language):

```
Printline(SymbolTable(8).Name + ' = ' + SymbolTable(8).Value)
```

READ accesses the symbol table by the index in op1 and reads the next integer in standard input into the *value* field of that entry.

## The Interpreter

The interpreter used for the Quad codes defined above represents a simple virtual machine. The features of the machine are:

- 1) it has one block of program memory, the quad structure, containing up to MAXQUAD (call it 1000) Quad rows, consisting of four integers per row: opcode, op1, op2, op3;
- 2) it has one block of data memory, the Symbol Table, structured as described above;
- 3) it has no registers;
- 4) it has a Program Counter, an integer, which always points to the next quad row to be executed;
- 5) it initializes its Program Counter to 0 when it starts, and after each quad row is executed, the Program Counter is incremented by 1 by default; a BRANCH of any kind is the only thing that can override this default incrementing behavior;
- 6) the machine stops when Program Counter = MAXQUAD or when the STOP opcode is encountered;

To simplify the interpretation process, and prevent the generation of more complex intermediate code and copy the symbol table to a separate file, and a number of other complications, **the interpreter must be coded in the language of the parser, and down the road, will be executed from the parser main program after a successful compilation has been accomplished.** It will need access to the symbol table entries by index number, and access to the quad codes by index number. The actual pseudocode framework appears below, in as generic and simple a form as practical:

```

Procedure Interpret; //PSEUDOCODE, not JAVA
VAR
    PC,                                //The program counter
    OPCODE, OP1, OP2, OP3 : integer;   //The current quad data

BEGIN
PC = 0; //assumes 0-based indexing on quad table
While (PC < MAXQUAD) DO
    BEGIN
        GETQUAD(PC, OPCODE, OP1, OP2, OP3); //the quad data at index PC
        IF Valid(OPCODE) //make sure case can handle it
            CASE OPCODE OF

                _DIV:
                    BEGIN
                        SymbolTable(OP3).Value := SymbolTable(OP1).Value /
                                                    SymbolTable(OP2).Value;

                        PC:= PC+1;
                    END;

//provide _MUL, _ADD, _SUB, _ASSIGN

                _JNZ:
                    BEGIN
                        IF (SymbolTable(OP1).Value <> 0) THEN
                            PC := OP3
                        ELSE
                            PC := PC+1;
                        END;

//provide all other branches, described above

                _STOP:
                    BEGIN
                        Printline('Execution terminated by program stop.');
```