CMPE 230

SYSTEMS PROGRAMMING

PROJECT 2

TRANSCOMPILER PROJECT

SERHAN ÇAKMAK

HALİL KARABACAK

30.04.2023

0- Summary Of The Previous Project

Previous advcalc project in C allows users to perform arithmetic operations on integers and variables.It supports many binary operations, as well as the ability to store and retrieve variables using a simple variable assignment syntax. The program prompts the user to enter an expression and then evaluates it using the order of operations. It also includes error handling for invalid inputs such as trying to calculate the value of an assignment including unmathced parantheses etc. With these features, it was a project that provides a useful tool for basic arithmetic calculations in C.

1- Introduction

In this project, we are expected to translate C code, which can calculate basic mathematical expressions along with type checking, to LLVM code. By assigning registers to both intermediate steps in the calculation and variables, we achieved connecting intermediate results and other calculation steps. After the syntax checking process, we print the necessary operations like alloc, store, load, and the other bitwise operations to the “X.ll” file if there is no syntactical error in the given input. If there is an error, then we just print to the terminal the lines that are erroneous.

2- Program Interface

In order to run the executable, the user just needs to call the make command with the input file that is wanted to be processed in the command line. Make command will create an executable and the executable will construct the “X.ll” file which consists of LLVM code. After creating the necessary files, the program will automatically be closed.

3- Program Structure

3.1) Infix To Postfix

Similar to previous homework, we again choose to use input to postfix process with a slightly different and in a more organized way. It also has a few additions.
Since I have explained infix to postfix before, I will just give a summary of what has been changed.

- We have made some changes to the infix to postfix conversion process. Now, we tokenize the input into different types such as OPEN_P, CLOSE_P, and MOD, which stand for opening parentheses, closing parentheses, and MOD operations, respectively. For instance, if we encounter the '%' symbol in the infix string, we add a node with the MOD type to our postfix stack. This way, we have a powerful tool that provides us with valuable information about the operations we are performing.

- Instead of storing the postfix expression in a string, we now use a Stack structure to store nodes, which makes our code more organized and easier to debug. This updated version

also handles modulo and division operators, in addition to the previous operators.

- The rest of the logic remains the same as it was in HW1, including parentheses handling and operator precedence.

3.2) Operate

One significant change has been made to the *operate* function. Instead of returning the result of the operation, it now calls a new function called fileWriter, which I will explain in more detail shortly. The rest of the logic remains the same as before. When it encounters a node with an operator type (such as ADD, MUL, or DIV), it pops two nodes from the stack. The operate function still holds the operator information, but now the operands do not have to be integers or variables; they can also be register values of LLVM IR. Without performing any further operations, the function calls fileWriter with the operands and operator.fileWriter function returns with a node that has REG type and adds it to the stack.

3.3) Creating The LLVM Code:

The new function, fileWriter, creates working LLVM IR code. It first determines the current operation based on the input operation type. For example, if the operation is addition (ADD), the corresponding LLVM IR operation is add. After deciding the operation, it creates a new node called final, which holds the result and has a type of REG. For simplicity, we have chosen to always return a REG-typed node in this function.

Once the result node is created, the function starts writing to the file using the fprintf built-in function of C. There are nine cases to consider, as the type of an operand node can be VAR, NUM, or REG. VAR represents a variable, NUM stands for decimal numbers,

and REG is used for register addresses of LLVM IR. We carefully handle all the cases since each one requires a different process. For example, if the node type is VAR, we need to load it into a REG in order to use it during the operation. Here's an example:

```
else if (a.type == VAR && b.type == REG) {
    fprintf(fpWrite, "    %%%d = load i32, i32* %%%s\n", regNum, a.regName);
    fprintf(fpWrite, "    %%%d = %s i32 %%%d, %%%s\n", regNum + 1, op, regNum, b.regName );
    regNum++;
}
```

In this case, since the type of node a is VAR, we first load it into a register and then perform the operation on the register. Another case is if the type is REG, in which case we need to write '%' before the address integer of the REG.

After generating LLVM IR code, we assign the register name of the final(result) node.

In short, this is the part of the code where we determine what to do with different operand type configurations and generate LLVM IR code accordingly.

4- Input-Output And Examples

There are some input and output files provided in order to display the sample usage of the program. One needs to remember that if there is a problem with the input provided .ll file will automatically be deleted and the erroneous line in the input will be printed to the terminal.

```
a = 7
bb = a * 5
c = (a - 2) * bb
d = ls (a, 3)
c
d
bb
e = xor (d, bb)
e
f = 0
g = lr(f, 1) * (0-a + 12)
h = rs(c, 2) | e
hh=0
not (xor (g, hh))
i = rr (g, 2) + not (xor (g, bb))
i
ls (h + 38, 1)
```

Sample input file

```
@print.str = constant [4 x i8] c"%d\0A\00"


define i32 @main() {
    %a = alloca i32
    store i32 7,i32* %a
    %1 = load i32, i32* %a
    %2 = mul i32 %1, 5
    %bb = alloca i32
    store i32 %2,i32* %bb
    %3 = load i32, i32* %a
    %4 = sub i32 %3, 2
    %5 = load i32, i32* %bb
    %6 = mul i32 %4, %5
    %c = alloca i32
    store i32 %6,i32* %c
    %7 = load i32, i32* %a
    %8 = shl i32 %7, 3
    %d = alloca i32
    store i32 %8,i32* %d
    %9 = load i32, i32* %c
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %9 )
    %11 = load i32, i32* %d
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %11 )
```

Sample LLVM code generated

5- Improvements And Extensions

In the code, improvements can be made in areas like variable checking (time complexity would be O(1) if the hashtable is implemented, in the current state the time complexity of traversing an array is O(N)) and error checking. Our code detects errors in an

iterative way on the string using characters, but tokenizing terms would be a better approach to check the syntactical errors.

6- Difficulties Encountered

The 3 most important problems we encountered on this project are allocating memory for variables, binding registers to the intermediate results, and not having some of the built-in functions in LLVM language.

A. Allocating memory for variables:
   The difficulty in allocating memory for variables is the need of checking whether the variable has been encountered before or not. According to the result of this check, we allocate a new memory and bind this memory to a new register. Calculations and the printing process is the first difficulty we faced.
B. Binding register to the intermediate results:
   Incrementing the global variable Regnum was one of the problems. We do assign a lot of registers to the variables and intermediate steps and correct incrementation of the Regnum variable was vital for us.
C. Not having some of the built-in functions in LLVM language:
   In LLVM language, there are no such things as bitwise rotation and bitwise not. So we had to code our own functions to appropriately call the shift, or, xor functions to achieve what rotation and not do.

Sample codes are provided to show the solution of the difficulties faced.

A)

```
// Assigning values to the variable names. If variable hasn't come before the allocate memory for it.
void writeAsssigment(char* var, StackNode value){
    if (lookup(var) ==0){
        fprintf( stream: fpWrite,  format: "    %%%s = alloca i32\n",var);
    }

    if (value.type == REG ){
        fprintf( stream: fpWrite,  format: "    store i32 %%%s,i32* %%%s\n", value.regName, var);
    }else if (value.type == VAR) {
        fprintf( stream: fpWrite,  format: "    %%%d = load i32, i32* %%%s\n", regNum, value.regName);
        fprintf( stream: fpWrite,  format: "    store i32 %%%d,i32* %%%s\n", regNum, var);regNum++;
    }
    else{
        fprintf( stream: fpWrite,  format: "    store i32 %d,i32* %%%s\n", value.value, var);
    }

}
```

B)

```
else if (a.type == VAR && b.type == REG) {
    fprintf( stream: fpWrite,  format: "    %%%d = load i32, i32* %%%s\n", regNum, a.regName);
    fprintf( stream: fpWrite,  format: "    %%%d = %s i32 %%%d, %%%s\n", regNum + 1, op, regNum, b.regName );
    regNum++;
}
```

C)

```
    }
    else if (operation == NOT){                    //Since not is an un
        StackNode notNode;                        // There is no direct
        notNode.type = NUM;
        *notNode.regName = '\0';
        notNode.value = -1;
        StackNode res = fileWriter(a,  b: notNode,  operation: XOR);
        return res;
    }
```

```
else if (operation == LR){                                          //
    StackNode first = fileWriter(a,b,  operation: LS );
    StackNode tmp;
    tmp.type = NUM;
    tmp.value = INT_BITS ;
    StackNode  sub = fileWriter( a: tmp,b,  operation: SUB);
    StackNode second = fileWriter(a,  b: sub,  operation: RSR);
    return fileWriter( a: first,  b: second,  operation: OR);
```

7- Conclusion

This project is prepared in order to teach how memory, CPU and other computer hardware work synchronously while the program is running by translating C code to an even lower level language than C, LLVM.