CMPE 230: SYSTEMS PROGRAMMING

01.05.2023

PROJECT 2

PREPARED BY:

- ERAY EROĞLU 2020400096
- BERKE KARTAL 2020400198

INTRODUCTION

The project that we are going to discuss is a transcompiler, a simple program that generates an LLVM IR format assembly code. The program reads a file, which is consisted of arithmetic operations, and writes the corresponding output to a file. The source code is written in C language.

The source code is implemented upon the previous project's source code. In the previous project, the main algorithm was creating a parsing tree corresponding to the current arithmetic operation. This is done by several parsing methods and the result is calculated by an evaluating method.

These parsing methods and the evaluating method are also used in the current project, with slight changes. The main idea behind the parsing tree is still the same, the given expression is parsed into terms and factors. After creating the tree, evaluating methods calculate the result by traversing the tree recursively. For further details about these methods, previous project documentation can be examined.

INPUT/OUTPUT

As a difference from the previous project, the transcompiler doesn't take inputs from the terminal as arithmetic operations. These operations should be stored in a file, and the program takes the path of this file as input. The corresponding output is written to another file with the same name but ".ll" extension.

However, this output file is generated as long as all the arithmetic operations in the input file are valid. There is an additional invalid input condition, compared to the previous project, which is the use of undefined variables. If the user tries to use a variable, which wasn't assigned any value yet, it is evaluated as an error. In case of an invalid input, an error message is printed to the terminal. The error message format is: "Error on line {line number}!".

IMPLEMENTATION

The program starts with reading the input file and storing it in a two-dimensional array. Then, arithmetic operations are handled similarly to the previous project, a single line is processed every iteration of while-loop. Unless there is an error in the current line, it will be evaluated and the same process will be done again until the end of the file.

In the previous project, we mentioned that expressions can be categorized into two different types: equations and non-equations. In case of non-equations, the program at first evaluates the expression, then calls a print function that was defined by us at the beginning of the output file.

In the case of equations, LHS must be a single variable and RHS must be an expression. Assuming the current line is a valid arithmetic operation, at first the transcompiler allocates space for the variable (32-bit integer) and writes the corresponding assembly code to the output file. (for a given line x = 5, allocation is done by this line of code: %x = alloca i32).

These variables must be stored for incoming operations. After the space allocation is done, the program evaluates the current line's result. This result must be assigned to the variable, this is done by the same method as the previous project: using a hash table. After the assignment, the transcompiler writes the assembly code for storing a variable, which is in this form(let's use the same example above, x = 5): store i32 5, i32* %x. To make this line of code work, space allocation for the variable must be done correctly.

After successfully allocating the space for a variable and storing its value, the next part is reaching its value when it is used in another arithmetic operation. This is done by the load function in assembly. In the parsing tree, when we reach a variable node, the program writes the load function to the output file, in assembly syntax (In our examples, we allocated space for x and stored its value, now let's reach its value: %1 = load i32, i32* %x). When we load a variable's value, we also need to store this operation in another variable. So, every loading statement is assigned a variable which is a number, starting with 1 and the number is increased by 1 after the statement is written.

There are built-in functions for every arithmetic operation in assembly, except not and rotate operations. These operations are implemented in the source code by chaining other operations. Not is implemented by using xor(expression, -1), however rotating functions are complicated. The main reason for the complication is the possibility of occurrences of constants, since they have their own values as integers, they aren't needed to be loaded. It would be more meaningful if the source code is examined directly, lines between (866 – 1010) (we didn't write it here since it is nearly 150 lines, which makes the documentation unnecessarily long)

CONCLUSION

This project helped us to understand the logic behind the assembly syntax. Also, improved our understanding of the concept of low-level programming languages. Since we are used to use C after the first project, writing this project was easier than the previous one. In addition to this, we think building the algorithm was also easier than the first project. To sum up, it was a good and beneficial experience.

EXAMPLES:

Input 1:

```
x = 5

y = x - 3

A = ls(y,2)

5 * A

b = not(A)

b + x
```

Output 1:

```
; ModuleID = 'advcalc2ir'

declare i32 @printf(i8*, ...)

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


define i32 @main() {

%x = alloca i32

store i32 5, i32* %x

%y = alloca i32

%1 = load i32, i32* %x

%2 = sub i32 %1,3

store i32 %2, i32* %y

%A = alloca i32

%3 = load i32, i32* %y

%4 = shl i32 %3,2

store i32 %4, i32* %A

%5 = load i32, i32* %A
%6 = mul i32 5,%5

call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %6)

%b = alloca i32

%8 = load i32, i32* %A

%9 = xor i32 %8,-1

store i32 %9, i32* %b

%10 = load i32, i32* %b

%11 = load i32, i32* %x

%12 = add i32 %10,%11

call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %12)

ret i32 0

}
```

Input 2:

```
x = lr(2,1)

y = x + 2 * 3 -5

mandalorian = ls(y,2)

bobafett = 5 * mandalorian

bobafett
```

Output 2:

```
; ModuleID = 'advcalc2ir'

declare i32 @printf(i8*, ...)

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


define i32 @main() {
```

```
%x = alloca i32

%1 = shl i32 2,1

%2 = sub i32 32,1

%3 = ashr i32 2,%2

%4 = or i32 %1,%3

store i32 %4, i32* %x

%y = alloca i32

%5 = load i32, i32* %x

%6 = mul i32 2,3

%7 = add i32 %5,%6

%8 = sub i32 %7,5

store i32 %8, i32* %y

%mandalorian = alloca i32

%9 = load i32, i32* %y

%10 = shl i32 %9,2

store i32 %10, i32* %mandalorian

%bobafett = alloca i32

%11 = load i32, i32* %mandalorian

%12 = mul i32 5,%11

store i32 %12, i32* %bobafett

%13 = load i32, i32* %bobafett

call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %13)

ret i32 0

}
```

Input 3:

```
7
5
54
6
87
```

Output 3:

```
; ModuleID = 'advcalc2ir'
declare i32 @printf(i8*, ...)
@print.str = constant [4 x i8] c"%d\0A\00"

define i32 @main() {
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 7)
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 5)
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 54)
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 6)
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 87)
ret i32 0
}
```