CmpE230 Project 2 Documentation

- *Transcompiler* -

Muhammet Ali Topcu - 2020400147

Abdullah Enes Güleş - 2021400135

01.05.2023

Purpose of the Project

Implementing an transpiler that translates input in the form of assignment statements and expressions of the AdvCalc++ language into LLVM IR code that can compute and output those statements.

Design

In this project, we first need to take the input as .adv files and create a .ll file to write the outputs. Then create tokens from the given input file with lexical analyzer. Then parse the tokens in order to create a parse tree (or abstract syntax tree) from the tokens. And finally, evaluate the ASTs one by one and write the equivalent LLVM IR code to .ll file.

How to Run the Program

1. Open a terminal at the root folder of the program.
2. Type 'make' to create an executable of the program. (make command runs the code in Makefile. It compiles the program and creates an executable named 'advcalc2ir.exe'
3. Then, with './advcalc2ir.exe file.adv' command, you can produce IR code in file.ll. Then 'lli file.ll' produces the output based on the given example code.

Implementation Details

The program runs mainly in main.c file.

Firstly, the program creates a .ll file with the same name as .adv file and writes the first couple of necessary lines of LLVM code to the .ll file.

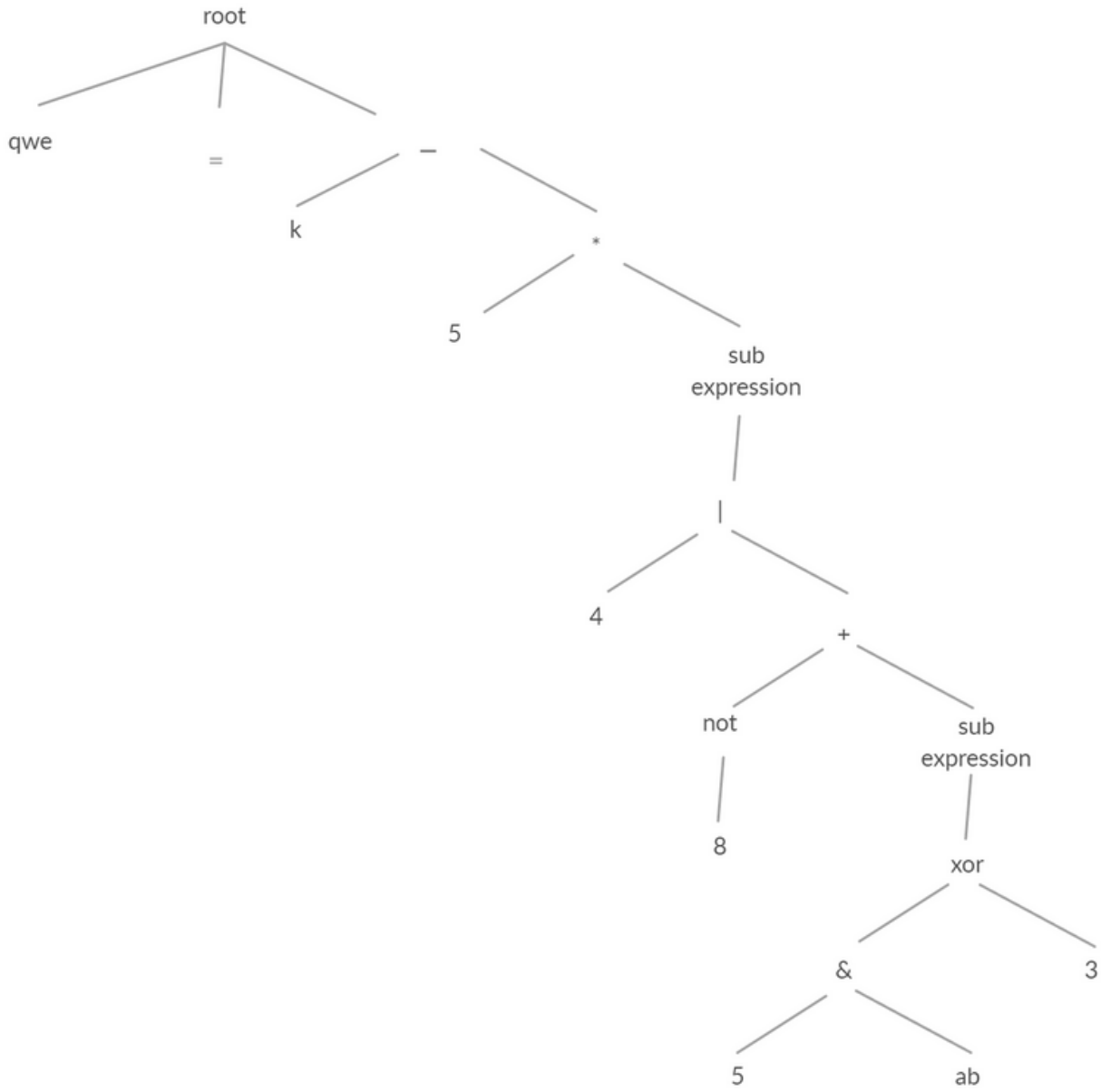
Then the program reads the .adv file line by line. Each line is stored in a temporary array and also an array of Tokens is created. A Token is a data type and defined at token.h header file to store the tokens properly.

Then, program proceeds and sends the temporary array and tokens array to lexer function, which can be found at lexer.c. The lexer function takes the input array and tokenizes the input. If an error occurs, the function returns 0 to notify that the input has an error and so that the program will not proceed anymore and continues to take the next input. If no error occurs, the function returns 1 and proceeds as expected.

After tokenizing the input, we need to create a parse tree to store the input in postfix format so that we can evaluate the postfix expression later with DFS.

At this stage, it was necessary to draw an example parse tree to see the operator precedences. So, we drew a parse tree to see mosst of the tokens in action.

In this example, the input was: **qwe = k - 5 * (4 | not(8) + xor(5 & ab, 3)))**



This input have covered most of the possible operator, function and parantheses precedences. And then, with the help of this example parse tree, we created a BNF to later use in parser functions:

root => <identifier > '=' <expr1> | <expr2>
expr1 => <expr2> | <expr1> '|' <expr2>
expr2 => <expr3> | <expr2> '&' <expr3>
expr3 => <expr4> | <expr3> '+' <expr4> | <expr3> '-' <expr4>
expr4 => <expr5> | <expr4> '*' <expr5> | <expr4> '/' <expr5> | <expr4> '%' <expr5>
expr5 => <expr6> | not(<expr1>) | xor(<expr1>,<expr1>) | ls(<expr1>,<expr1>) | rs(<expr1>,<expr1>) | lr(<expr1>,<expr1>) | rr(<expr1>,<expr1>)
expr6 => <expr7> | '(' <expr1> ')'
expr7 => identifier | constant

Using this BNF, we coded the parser functions. At the main function, after tokenizing the input with lexer function, program creates root nodes for ASTs. We implemented AST in ast.c to store the inputs in postfix format. At this stage, program has two possibilities: Either input is an expression or an assignment.

In case of an assignment, parser function takes the tokens and creates the tree and returns it. If there exists a problem, it returns null. Otherwise, it returns a result tree. Since this is an assignment, we need to store the identifier. So, we also implemented allocated_identifiers array to allocate each identifier once at the beginning of the LLVM IR code.

In case of an expression, parser function takes the tokens and creates a tree and returns it, same as the above example.

After all the root nodes are created and stored in an array called roots, we will evaluate the ASTs one by one. Evaluation of the trees is implemented in evaluator.c with eval function. This function takes the root of a tree, traverses the tree recursively and writes the equivalent LLVM IR code to .ll file. At this stage, the program uses identifiers_list array to check whether an identifier is called before it is declared.

Difficulties Encountered

We faced difficulties again during memory management. We also had a similar problem while implementing advcalc but this time we learned from our previous experience and we easily figured out that it was necessary to use malloc for dynamic memory allocation.

The other problem that we encountered was creating the operations that were not in LLVM as single methods. These operations are left rotate, right rotate, not, etc. We were trying to find a single method that exists in LLVM but there was none. So we had to create a set of lines to able to do those operations.

To see the other difficulties that we encountered you can check our previous documentation of advcalc.

Example Inputs & Outputs

Example 1

```
x=3
y=5
zvalue=23+x*(1+y)
zvalue
k=x-y-zvalue
k=x+3*y*(1*(2+5))
k + 1
```

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

define i32 @main() {
    %x = alloca i32
    %y = alloca i32
    %zvalue = alloca i32
    %k = alloca i32
    store i32 3, i32* %x
    store i32 5, i32* %y
    %x0 = load i32, i32* %x
    %x1 = load i32, i32* %y
    %x2 = add i32 1, %x1
    %x3 = mul i32 %x0, %x2
    %x4 = add i32 23, %x3
    store i32 %x4, i32* %zvalue
    %x5 = load i32, i32* %zvalue
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x5)
    %x6 = load i32, i32* %x
    %x7 = load i32, i32* %y
    %x8 = sub i32 %x6, %x7
    %x9 = load i32, i32* %zvalue
    %x10 = sub i32 %x8, %x9
    store i32 %x10, i32* %k
    %x11 = load i32, i32* %x
    %x12 = load i32, i32* %y
    %x13 = mul i32 3, %x12
    %x14 = add i32 2, 5
    %x15 = mul i32 1, %x14
    %x16 = mul i32 %x13, %x15
    %x17 = add i32 %x11, %x16
    store i32 %x17, i32* %k
    %x18 = load i32, i32* %k
    %x19 = add i32 %x18, 1
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x19)
    ret i32 0
}
```

```
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ make
gcc -o advcalc2ir.exe main.c -I.
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ ./advcalc2ir.exe file.adv
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ lli file.ll
41
109
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ llc file.ll -o file.s
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ clang file.s -o myexec
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ ./myexec
41
109
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$
```

Example 2

```
siu = 11
siuuu = 7
siu / siuuu
siu = siu * siuuu
siu - siu + siu * siu / siu
```

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

define i32 @main() {
    %siu = alloca i32
    %siuuu = alloca i32
    store i32 11, i32* %siu
    store i32 7, i32* %siuuu
    %x0 = load i32, i32* %siu
    %x1 = load i32, i32* %siuuu
    %x2 = sdiv i32 %x0, %x1
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x2)
    %x3 = load i32, i32* %siu
    %x4 = load i32, i32* %siuuu
    %x5 = mul i32 %x3, %x4
    store i32 %x5, i32* %siu
    %x6 = load i32, i32* %siu
    %x7 = load i32, i32* %siu
    %x8 = sub i32 %x6, %x7
    %x9 = load i32, i32* %siu
    %x10 = load i32, i32* %siu
    %x11 = mul i32 %x9, %x10
    %x12 = load i32, i32* %siu
    %x13 = sdiv i32 %x11, %x12
    %x14 = add i32 %x8, %x13
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x14)
    ret i32 0
}
```

```
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ make
gcc -o advcalc2ir.exe main.c -I.
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ ./advcalc2ir.exe file.adv
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ lli file.ll
1
77
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ llc file.ll -o file.s
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ clang file.s -o myexec
^[[Aalitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ ./myexec
1
77
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$
```

Example 3

```
a = 8
b = 8 * (a - 6)
c = b + 6
ls(a, 2)
xor(b - a + 1, 17)
y = 2 * b - c
xor(rs(y, 2), not(c))
```

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

define i32 @main() {
    %a = alloca i32
    %b = alloca i32
    %c = alloca i32
    %y = alloca i32
    store i32 8, i32* %a
    %x0 = load i32, i32* %a
    %x1 = sub i32 %x0, 6
    %x2 = mul i32 8, %x1
    store i32 %x2, i32* %b
    %x3 = load i32, i32* %b
    %x4 = add i32 %x3, 6
    store i32 %x4, i32* %c
    %x5 = load i32, i32* %a
    %x6 = shl i32 %x5, 2
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x6)
    %x7 = load i32, i32* %b
    %x8 = load i32, i32* %a
    %x9 = sub i32 %x7, %x8
    %x10 = add i32 %x9, 1
    %x11 = xor i32 %x10, 17
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x11)
    %x12 = load i32, i32* %b
    %x13 = mul i32 2, %x12
    %x14 = load i32, i32* %c
    %x15 = sub i32 %x13, %x14
    store i32 %x15, i32* %y
    %x16 = load i32, i32* %y
    %x17 = ashr i32 %x16, 2
    %x18 = load i32, i32* %c
    %x19 = xor i32 %x18, -1
    %x20 = xor i32 %x17, %x19
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x20)
    ret i32 0
}
```

```
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ make
gcc -o advcalc2ir.exe main.c -I.
^[[Aalitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ ./advcalc2ir.exe file.adv
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ lli file.ll
32
24
-21
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ llc file.ll -o file.s
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ clang file.s -o myexec
^[[Aalitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ ./myexec
32
24
-21
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$
```

Example 4

```
x = 1
y = x + 3
z = x * y * y*y
z
xor(((x)), x)
xor(((x)), x) | z + y
rs(xor(((x)), x) | z + y, 1)
ls(rs(xor(((x)), x) | z + y, 1), (((1))))
```

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

define i32 @main() {
    %x = alloca i32
    %y = alloca i32
    %z = alloca i32
    store i32 1, i32* %x
    %x0 = load i32, i32* %x
    %x1 = add i32 %x0, 3
    store i32 %x1, i32* %y
    %x2 = load i32, i32* %x
    %x3 = load i32, i32* %y
    %x4 = mul i32 %x2, %x3
    %x5 = load i32, i32* %y
    %x6 = mul i32 %x4, %x5
    %x7 = load i32, i32* %y
    %x8 = mul i32 %x6, %x7
    store i32 %x8, i32* %z
    %x9 = load i32, i32* %z
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x9)
    %x10 = load i32, i32* %x
    %x11 = load i32, i32* %x
    %x12 = xor i32 %x10, %x11
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x12)
    %x13 = load i32, i32* %x
    %x14 = load i32, i32* %x
    %x15 = xor i32 %x13, %x14
    %x16 = load i32, i32* %z
    %x17 = load i32, i32* %y
    %x18 = add i32 %x16, %x17
    %x19 = or i32 %x15, %x18
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x19)
    %x20 = load i32, i32* %x
    %x21 = load i32, i32* %x
    %x22 = xor i32 %x20, %x21
    %x23 = load i32, i32* %z
    %x24 = load i32, i32* %y
    %x25 = add i32 %x23, %x24
    %x26 = or i32 %x22, %x25
    %x27 = ashr i32 %x26, 1
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x27)
    %x28 = load i32, i32* %x
    %x29 = load i32, i32* %x
    %x30 = xor i32 %x28, %x29
    %x31 = load i32, i32* %z
    %x32 = load i32, i32* %y
    %x33 = add i32 %x31, %x32
    %x34 = or i32 %x30, %x33
    %x35 = ashr i32 %x34, 1
    %x36 = shl i32 %x35, 1
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %x36)
    ret i32 0
}
```

```
^[[Aalitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ ./advcalc2ir.exe file.adv
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ lli file.ll
64
0
68
34
68
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ llc file.ll -o file.s
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ clang file.s -o myexec
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$ ./myexec
64
0
68
34
68
alitpc25@DESKTOP-QG65TQ3:~/cmpe230project2son$
```