Course: CMPE 230 SYSTEMS PROGRAMMING

Students names: GÜLSEN SABAK - MAHMUT BUGRA MERT

Submitted Person: MAHMUT BUGRA MERT

Project Name: TransCompiler

PROGRAMMING PROJECT

Submission Date: 01.05.2023

Phase 1: INTRODUCTION:

In this project, we developed a 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. Our first algorithm is based on the lexical analysis and Shunting Yard algorithm which helps us to transform infix expressions and assignments to postfix notation. While implementing our first algorithm, we used different data structures. These data structures are: Linked List based Stack and Queue to implement Shunting Yard algorithm and Dictionary to calculate assignments such as "a = 5 +3" etc. Also, we determine precedence orders among operators and functions for the calculation part to calculate the values written in the interpreter correctly. Our second algorithm to create the intermediate language is making an llvm file using modified version of the Shunting Yard algorithm. After we create the llvm file, we execute the file and get the result. While implementing the intermediate language, we used a built-in “fprintf” function and counter for the number of calls. For example, we use counter for the number after % symbol:

```
 %1 = load i32, i32* %x 13
%2 = load i32, i32* %y 14
 %3 = add i32 1,%2.
```

Also we changed an operator which is % operator. In the first Advcalc project we use % as a comment symbol, but in the Transcompiler project we use % as a modulo operator. In addition, we add a division operator to use it in the Transcompiler. Moreover, we control if there is an undefined identifier based error or not in 2 ways. One of them is controlling if there is one Token and if its type is an identifier. The other is controlling whether an identifier is in the dictionary or not. Lastly, we create an output file in llvm form and we print the result which is calculated from the llvm file, to the terminal.

Phase 2: PROGRAM INTERFACE:

To run the program, first you need to open the terminal. Then, you should go to the root folder where "project2.c" and "makefile" are located via terminal. After that, you can run the program by typing "make" and pressing enter. As a result, an executable named “advcalc2ir” will be created.

Phase 3: PROGRAM EXECUTION:

For the input part, we expect to get an input file as an argv[1]. In the input file, we expect expressions and assignments. Also, the numbers have to be in "integer" format. In addition, we expect to calculate "summation (a + b), multiplication (a * b), subtraction (a - b), division (a / b), modulo (a % b), bitwise and (a & b), bitwise or (a | b), shifting right (rs(a,i)) and left (ls(a,i)) , rotating right(rr(a,i)) and left (lr(a,i)), bitwise xor (xor(a,b)), and bitwise complement (not(a))" and print the correct result to the terminal. Moreover, we expect to create a llvm version of the input file when we read the file. Also, Transcompiler helps users to assign a number to a variable, and update this variable later. However, if you use a variable directly without declaring it, you will get an error in the llvm file such as “Error on line 8!”. Also, the Transcompiler is

case-sensitive for variable names. For the output part, we have 2 options which are printing the result, and printing the error message.
Case for printing result: If all operations are valid and all numbers are integer, then the Transcompiler creates the intermediate language to the output file and calculates the result and prints it.
Case for Errors: There are plenty of edge cases to print error message but I will give below, some cases for printing Error:
If parentheses are in the wrong order or missing, then we print the error message.
If there is an undefined variable in a line in the input file, then we print the error message to the terminal and at the end of the input file we delete the llvm file.
If a user puts an "n" parameter inside the function which expects "m" parameters (n≠m) , then it prints an error message.
At the end, the program will stop when the reading of the file finishes.

Phase 4: INPUT AND OUTPUT:

The Transcompiler program takes input as a file from argv[1]. Output is the intermediate version of the input file. If there is no error, then the llvm file will be created. Also, if the result can be generatable, which means there is no error, then the result will be seen in the terminal. However, if there is an error in the input file, then the generated llvm file will be deleted after all input file lines are read, and errors will be printed to the terminal with their line number.

Normally there are no numbers at the beginning of the input lines. I put numbers to show that error-line matching is true.
For example in "a)" part output will be printed.

```
a)
 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))))
```

Output:
64
0
68
34
68
However, in "b)" part output will not print the result, it will just print the errors.

b)
1) xor (a, b + 3)
2) x or (a -b, b + 3)
3) (a - b) = 3
4)
5)
6)))((a))

Output:
Error on line 1!
Error on line 2!
Error on line 3!
Error on line 6!

Phase 5: PROGRAM STRUCTURE:
Data Types:
Token: Token is a single unit we get, after parsing the line. It has 3 data fields.
Data Fields:
`TokenType type` –> Stores the type of the token.
For example: TOKEN_TYPE_NUMBER
`char val[257]` –> Token value. For example: "xor", "24", "a", "(" .
`char identifierValue[257]` –> Identifiers' integer value. For example: "10", "0", "24".

TokenType: It is an enumerator. It holds Token types.

Node: Node has 2 different properties. One of them is Token and the other is Node pointer which points to the next Node. It is used for implementing Queue, Stack, and Dictionary.
Data Fields:
`Token data` –> Stores Tokens.
`Node* nextNode` –> Points to the next Node.

Queue Implementation: Queue is used to implement Shunting-yard Algorithm. It has 4 functions:
1) **`void enqueue(Node* *firstNode, Token data):`** It creates a Node, stores a Token(data) in it, and bonds the Node with the other Nodes. "firstNode" is a pointer to the pointer which points to the first Node of the Queue.

2) **`void dequeue(Node* *firstNode):`** It deletes an element from the end of the Queue. "firstNode" is a pointer to the pointer which points to the first Node of the Queue.

3) **`void printQueue(Node* currentNode):`** We use it to debug the code, but it is not used intrinsically. If you want to understand the algorithm correctly, you can call it in the main part. It prints the elements of the Queue. "currentNode" is a pointer to the first Node of the Queue.
4) **`bool hasOneElt(Node* currentNode):`** It returns true if the Queue has one element. "currentNode" is a pointer to the first Node of the Queue.

Stack Implementation: Stack is used to implement Shunting-yard Algorithm. It has 5 functions:
1) **`void push(Node* *firstNode, Token data):`** It creates a Node, stores a Token(data) in it, and bonds the Node with the other Nodes. "firstNode" is a pointer to the pointer which points to the first Node of the Stack.

2) **`Token peek(Node* *firstNode):`** It returns the element at the top of the Stack. "firstNode" is a pointer to the pointer which points to the first Node of the Stack.

3) **`void pop(Node* *firstNode):`** It deletes the element at the top of the Stack. "firstNode" is a pointer to the pointer which points to the first Node of the Stack.

4) **`bool isEmpty(Node* firstNode):`** It returns true if the Stack is empty. "firstNode" is a pointer to the first Node of the Stack.

5) **`void printStack(Node* currentNode):`** It prints the elements of the Stack. "currentNode" is a pointer to the first Node of the Stack.

Dictionary Implementation: Dictionary is used to store initialized variables. It has 5 functions:
1) **`char* get(Node* startNode ,char* identifier):`** It returns the identifierValue of the given identifier. "startNode" is a pointer to the first Node of the Dictionary.

2) **`void add(Node* *startNode, Token newToken):`** It adds a new identifier to the Dictionary. "startNode" is a pointer to the pointer of the first Node of the Dictionary.
3) void printDict(Node* startNode): It prints the elements of the Dictionary. "startNode" is a pointer to the first Node of the Dictionary.

4) **`void updateValue(Node* startNode, char newValue[257], char identifier[257] ):`** It updates the value of an identifier in the Dictionary. "startNode" is a pointer to the first Node of the Dictionary.

5) **`bool doesExist(Node* startNode ,char* identifier):`** It returns true if the given identifier exists in the Dictionary. "startNode" is a pointer to the first Node of the Dictionary.

We can summarize our code in 3 main steps:
- First, make lexical analysis of the input line with parse(),

- Then, transform the analyzed line from infix to postfix notation.
- Then, create a llvm file for the intermediate language. If there is no error, llvm file (output file) will calculate the result and print it to the terminal. If there is an error, then close the llvm file and print the error to the terminal in the form: “Error on line 8!”.

Phase 6: EXAMPLES:

Example in the below shows some error types in Transcompiler.
For example,
In line 7, the reason of the error is Unary operation
In line 8, the reason of the error is using operators wrongly
In line 9, the reason of the error is using functions wrongly (missing variable and comma)
In line 10, the reason of the error is missing parentheses
In line 11, the reason of the error is undefined variable (maeni)

1) Kuzurete = 99
2) Yuku = 75
3) Maeni = 12
4) Kuzurete + Yuku | 1
5) Maeni * Kuzurete | 1
6) Yuku - Maeni | 1
7) - Kuzurete
8) & Yuku & 1
9) xor(Maeni &)1
10) xor(Kuzurete * Yuku * Maeni, 0
11) xor(Kuzurete * Yuku * maeni, 0)

Output:
Error on line 7!
Error on line 8!
Error on line 9!
Error on line 10!
Error on line 11!

Example in the below, shows what happens when the input file is correct. Calculation done and printed to the terminal.
Input:
a = 7
bb = a * 5
c = (a - 2) * bb
d = ls (a, 3)

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

Output:

```
175
56
35
27
-1
-36
194
```

LLVM version of the code:

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

define i32 @main() {
        %a = alloca i32
        store i32 7, i32* %a
        %bb = alloca i32
        %1 = load i32, i32* %a
        %2 = mul i32 %1,5
        store i32 %2, i32* %bb
        %c = alloca i32
        %3 = load i32, i32* %a
        %4 = sub i32 %3,2
        %5 = load i32, i32* %bb
        %6 = mul i32 %4,%5
        store i32 %6, i32* %c
        %d = alloca i32
        %7 = load i32, i32* %a
        %8 = shl i32 %7,3
```

```
	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 )
	%13 = load i32, i32* %bb
	call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32
%13 )
	%e = alloca i32
	%15 = load i32, i32* %bb
	%16 = load i32, i32* %d
	%17 = xor i32 %15,%16
	store i32 %17, i32* %e
	%18 = load i32, i32* %e
	call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32
%18 )
	%g = alloca i32
	%20 = sub i32 32,1
	%21 = shl i32 0,1
	%22 = lshr i32 0,%20
	%23 = or i32 %21,%22
	%24 = load i32, i32* %a
	%25 = sub i32 0,%24
	%26 = add i32 %25,12
	%27 = mul i32 %23,%26
	store i32 %27, i32* %g
	%h = alloca i32
	%28 = load i32, i32* %c
	%29 = ashr i32 %28,2
	%30 = load i32, i32* %e
	%31 = or i32 %30,%29
	store i32 %31, i32* %h
	%32 = load i32, i32* %g
	%33 = xor i32 0,%32
	%34 = xor i32 %33,-1
	call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32
%34 )
	%i = alloca i32
	%36 = load i32, i32* %g
	%37 = sub i32 32,2
	%38 = lshr i32 %36,2
	%39 = shl i32 %36,%37
```

```
        %40 = or i32 %38,%39
        %41 = load i32, i32* %bb
        %42 = load i32, i32* %g
        %43 = xor i32 %41,%42
        %44 = xor i32 %43,-1
        %45 = add i32 %40,%44
        store i32 %45, i32* %i
        %46 = load i32, i32* %i
        call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32
%46 )
        %48 = load i32, i32* %h
        %49 = add i32 %48,38
        %50 = shl i32 %49,1
        call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32
%50 )
        ret i32 0

}
```

Phase 7: IMPROVEMENTS AND EXTENSIONS:

Actually, in our perspective there is a weak point about the Transcompiler. For example, we cannot add comments to any line. Apart from this, the Transcompiler can just calculate, if the inputs are integer. We can optimize it to calculate float numbers too. Also, the Transcompiler can just calculate the results if there is no error in the input file. In our perspective, we can optimize the Transcompiler, to calculate the results even if there are some errors in the input file. In our opinion, one of the strong points about advcalc is, it can calculate some different expressions which normal calculators cannot do. For example it can calculate left, right rotations and left, right shifts .

Phase 8: DIFFICULTIES ENCOUNTERED:

Actually, we haven't faced much difficulty when we are implementing the project. However, sometimes we have trouble handling error mechanisms. We tried to implement a couple of solutions to handle errors, and the last solution we tried was okay. Our solution is that we created a boolean variable(isError) which is true when there is an error in the line and false when there is no error in the line. When isError is true, we broke the calculation loop, printed an error message to the terminal, and moved to the next line of input.

Phase 9: CONCLUSION:

At the end of this project, most importantly, right now we can understand the process behind programming languages. We have learnt how to convert the code to the intermediate language.

Also, we have learnt how to use struct in C and create our data types, using pointers effectively, using the past knowledge for creating Queue, Stack, Dictionary, working as a team, general structures of llvm form and reading file in C. In our perspective, this project is easier than the first project with respect to implementation. In addition, it is very appropriate to understand how systems work in a computer deeply. In a nutshell, implementing a Transcompiler is a little bit hard, but joyful and teachful.