AdvCalc++ Transcompiler Documentation

Ömer Faruk Erzurumluoğlu & Yusuf Kağan Çiçekdağ

April 2023

1 Description Of The Project

The aim of the project is to write a transcompiler program in C programming language, which takes an input file from the user. In case of a valid input, it creates an LLVM IR code file which can assign values to variables and calculate some basic expressions. Otherwise, it prints “Error on line x” (where x denotes line number where an error occurs) for all lines which contain errors.

2 Rules & Restrictions

There are some basic rules in order to use the program properly:

- LLVM IR uses static single assignment (SSA) based representation. In assignment statements, variables are assigned a value once.
- alloca is used to allocate space for variables and return address of allocation.
- Variables start with the % sign.
- The i8, i16, and i32 keywords mean 8 bit, 16 bit, and 32 bit types, respectively
- The * character denotes a pointer, just as in C.
- The IR contains the following piece of code, which defines the module name and the prototype for the printf function. You should use & generate this part as is in your translated IR files:

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

- To print the value of a variable you can can use the printf function we’ve defined above.

```
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* \
@print.str, i32 0, i32 0), i32 %7 )
```

- Every value and every calculation will be integer-valued (divisions should be rounded as it is done in C - i.e. 8 / 3 should be equal to 2).
- There will be no expressions or assignments with a result that exceeds a 32-bit number.
- Similarly, every bit-wise intermediate operation will abide by the 32-bit limit.
- The language does not support the unary minus (-) operator (i.e x = -5 or a = -b is not valid). However, as can be seen above, the subtraction operation is allowed.
- The variable names will consist of lowercase and uppercase Latin characters in the English alphabet [a-zA-Z].
- Expressions or assignments will consist of 256 characters at most.
- The input advcalc++ language may include all sorts of syntax errors, and all of them should be dealt with.
- Undefined variables cause an error in advcalc++.

- In case of syntax errors or undefined variables in the files, your output(s) should report them to the terminal in following form:

Error on line 8!
Error on line 13!

- All of the following can be given as an input - and all of them are valid: a + b, a+ b, a + b, a+b, a +b, (((a))) + (b)
- The valid operations are + (addition) , - (subtraction) , * (multiplication) , & (and operator) , | (or operator) , / (integer division operator) and % (modulo operator).
- The valid functions are xor, ls, rs, lr, rr and not functions.
- There will be no comments inside the code

---

3 How To Use The Program

The input file will be given as an argument during the running of our program. Inside the file, one can either declare an assignment to a new or pre-used variable and assign it the result of the right hand side expression. Alternatively, he/she can just write an expression and the program will create the LLVM IR code which evaluates the given expression and prints its result After reaching the end of the input file, the program terminates. If there exists an error (or errors), no output file is created and all line numbers where an error occurs are printed with their numbers. Otherwise, the LLVM IR code file which executes the given expressions is created and this code is returned as an output file.

---

4 Implementation

Since this is a transcompiler, we wanted it to be fast. Thus, it only iterates once through the input in a recursive manner. Instead of tokenizing the input, we evaluate by using EBNF. For each non terminal in our EBNF, we have a function for its rule. As an example let's say our input is
'a= 5+not(1)' then our code will call the necessary functions as explained below:

```
solveStatement:
   getVar;
   solveExpr:
      solveOp( precedence = 1):
         solveOp( precedence = 2):
            solveOp( precedence = 3): ( Since the precendence of '+' is 3, it will solve its left and right sides first )
               solveOp( precedence = 4): (left side of '+')
                  solveOp( precedence = 5):
                     getString.
               solveOp( precedence = 4): (right side of '+')
                  solveOp( precedence = 5):
                     getString:
                        getVar; (for finding function's name)
                        solveExpr: (first solves the inside of the function)
                           solveOp( precedence = 1):
                              solveOp( precedence = 2):
                                 solveOp( precedence = 3):
                                    solveOp( precedence = 4):
                                       solveOp( precedence = 5):
                                          getString.
```

---

5 Dependencies

Our program has its dependant libraries stated below:

- stdio.h: The standard library for input/outputs.
- stdlib.h: The standard library that provides a set of general-purpose functions. Inside our program, we've used it for memory allocation purposes
- string.h: The standard library for string operations
- ctype.h: A standard library which declares several functions that are useful for testing and mapping characters.
- hashmap.h: Our custom library for HashMaps
- standart.h: Our custom library, used for defining booleans
- deneme.h: Our header for the functions used in the main code (deneme.c)
- stack.h: Our header for the functions used for stacks inside the main code (deneme.c)

---

6 Global Variables

- char mainStatement[257]: The read line inside the main function. It is given that any line can have at most 256 characters long, so our mainStatement array has 257 elements, as we also include "\n" as a character as well.
- int counter: An integer variable used as an iterator for iterating through our line (initialized as 0).
- char deflastChar[2]: Our default last characters for a string (initialized as '\n' , 0).
- char lastforparanthesis[3]: Last characters for expressions inside parenthesis (initialized as '\n' , ')', 0).
- char lastforfunc[3]: Last characters for expressions inside functions (initialized as '\n' , ' , ', 0).
- bool8 error: A boolean flag to indicate if there were any errors. (initialized as FALSE, in case of a incorrect line, this turns into TRUE during the running of the code). This boolean checks whether the read line has errors or not.
- HashMap variables: A hashmap for storing the variables with their values.
- char *functions[6]: Array of function names (initialized as "xor", "ls", "rs", "lr", "rr", "not")
- FILE *output: Pointer to our output file.
- int linecounter = 1: Counts which line we are reading.
- int variableCounter = 0: Counts how many variables we've used so far during the LLVM IR code translation.
- bool8 areThereAnyErrors = FALSE: Checks whether there exist any lines with errors. If no error occurs, it remains FALSE, otherwise it returns to TRUE.

6.1: Stack-Related Global Variables

- char* stack[1000]: The stack for stored variables and numbers, which are used inside the writing of LLVM IR code.
- int stackCounter = 0: The number of elements inside the stack above, initialize as 0, as there are no elements inside the stack initially.
- char* irstack[1000]: The stack for instructions (but in a reversed order), which are used inside the writing of LLVM IR code. If the instructions are going to be printed, in order to arrange the order, the instructions inside are sent to the istack,

and then will be written to our LLVM IR code.

- int irstackCounter = 0: The number of elements inside the stack above, initialize as 0, as there are no elements inside the stack initially.

---

7 Newly Defined Structures

Inside the program, there are three defined structures, which are explained below:

7.1 HashMapEntry

This struct is for storing the entries of our HashMap used inside the program.
Its fields are:

- char *key: The string (stored as a char pointer) key of the entry
- int value: The (64-bit) integer value for the entry

7.2 HashMap

This struct is for storing the entries, where the entries' key is the assigned variable and value is the integer which the variable is assigned to. Its fields are:

- HashMapEntry *array: The array used for storing the entries, stored as a pointer which holds an array of HashMapEntry s
- int fullness: The integer denoting how many elements are currently inside our HashMap
- int size: The integer denoting the current size of the HashMap

Note that inside the hashmap.c file, we also initialized some factors for the HashMap, which are:

- hashhap min size = 257: When a new HashMap is constructed, this will be the initial size of our HashMap
- max load = 0.5f: The initialized load factor for resizing the HashMap
- size factor = 2: The initialized resizing factor for the HashMap

---

8 Functions

The functions used in the project are divided into 3 parts:

1. HashMap Functions
2. Main Code Functions
3. Stack Functions

8.1 HashMap Functions

- HashMap initializeHashMap(): This function creates a HashMap, initializes its fields and returns the newly created HashMap
- int hash(char *str, int size): This is our primary hashing function used inside our HashMap

- int doubleHash(char *str, HashMap *hashMap): In case of a collision due to first hashing function, this hashing function will be used until a place is found
- void biggerarray(HashMap *hashMap, int newSize): If our HashMap's load factor is exceeded, this function will increase the size of our HashMap by size factor (in this program, load factor is initialized as 0.5 and size factor is initialized as 2)
- void add new element(HashMap *hashMap, char *key, int value): Function for adding a new element to our HashMap
- int getValue(HashMap hashMap, char *str): Function getting the value of a given key (str)
- void deconstractor(HashMap *hashMap): Frees the pre-allocated memories of our HashMap
- int chadstrcmp(char *str1, char *str2): String comparator function
- bool8 isAllocated(Hashmap *hashMap, char *variable) : Returns whether a variable has already been allocated inside the HashMap or not.

8.2 Main Code Functions

- int main(): This is our main function. It takes inputs and processes them until the termination of the program
- void solveStatement(): Checks whether the given input is an assignment or a statement and proceeds accordingly.
- int getString(): Starting from the counter, this function gets the nearest string, solves it, then returns its value, where a "string" can be an expression inside parentheses, a variable, a function with two given expressions inside, a not function with given expression inside, or just a number.
- int solveExpr(char *last): Solves the expression starting from the counter until it reaches a last character.
- int getOper(char op): Gets the operation, and returns its precedence number
- void getVar(char *assignedvar): Gets the nearest variable starting from the counter
- void dismissblank(): Skips the empty spaces
- int solveOp(int precedence, char *last) : Solves the operations in a recursive manner, considering precedences as well.
- void initialize sides(int *left, int *right): Gets the first and second parameters of a function
- char* getNewVar() : Creates and returns us %(variableCounter) as a string
- void toString(char* str, int num) : Takes the input num, and creates its string version, which will be str.

8.3 Stack Functions

- void push(char *str): Pushes the value str into the stack, increments the stackCounter by 1.
- char* pop() : decrements stackCounter by 1, returns the popped value from stack
- void irpush(char *str): Pushes the value str into the irstack, increments the irstackCounter by 1.

- char* irpop() : decrements irstackCounter by 1, returns the popped value from irstack
- void drain() : Drains the elements of irstack (which are our LLVM IR instructions) and frees them.
- void writeAllInstructions(FILE *file) : Writes the elements (instructions) inside the istack to the given file (which will be our output file inside our code) by popping them one by one.

9 Algorithm

1- Read the given input file, read each line one by one. Create an output file for writing the required LLVM IR commands, and write the beginning part of the file.

2- Apply solveStatement function for each line. After reading the entire file, write the ending part of the LLVM IR code to the output file. If there were any errors inside the given input file, delete the output file, and report the lines with errors by printing “Error on line x!” onto the terminal, where x takes all possible line numbers which an error occurs.

3- Inside solveStatement, get the nearest variable (if exists) by getVar function, and skip redundant empty spaces.

4- If it is an expression, set counter to 0 and start solveExpr and if no such error occurs, write the LLVM IR code which prints the result, else print “Error on line x!” where x is the number of line that we are reading.

5- If it is an assignment, solve the right hand side by setting counter into the starting point of the expression and start solveExpr function, if no such error occurs, write the LLVM IR code which stores the variable name-variable value pair, else print “Error on line x!” where x is the number of line that we are reading.

6- Inside solveExpr, we just call solveOp(precedence = 1) and return its result. Note that solveOp and getString functions deals with possible errors.

7- Inside getString, if we see an open parenthesis, we solve part until the closing parenthesis (of the parenthesis we’ve seen) by solveExpr and write the LLVM IR code which assigns its value to a variable. If we see a function name, we solve the inside part of the function, and write the LLVM IR code which assigns its value to a variable. If we see a variable, we get the corresponding number from the “variables” hashmap and write the LLVM IR code which loads its stored value to that variable. If we see a number, we solve the number itself (actually, we just read the number and write the LLVM IR code which takes that number by a variable). During these cases, errors might occur as well. In such a case, we make the error flag “TRUE”.

8- Inside solveOp(precedence), we have a numerical expression, and we start reading the expression. If an error is found, return 0 and terminate the function. Else, we first read the left part by solveOp(precendence+1) function and skip redundant empty spaces in order to get the right part. If we see an operator with given precedence, we divide the expression into two parts, and while the second part is divisible, we keep dividing the expression into parts seperated by operators with the given precedence in a similar manner. Then, we solve each part by solveOp(precendence+1) and combine their results, while also writing the LLVM IR code which evaluates the results and does the necessary operations. In case of an error, we turn the error flag to TRUE. Note that solveOp(precedence) stops at the given last characters and when it sees operations with other precedences. Also, as there are no operations having more precedence than *,/ and % (they have precedence=4), precedence=5 case means that we cannot seperate the expression into smaller parts, so we just take the numbers.

10 Some Possible Errors and Their Solutions

We used a BNF to determine if a statement is consistent or not. The program will not look at redundant whitespaces. Everything else is an error for the program. Here is our EBNF:

- <statement> := (<assignment> |<expression>)
- <assignment> := <var> = <expression>
- <expression> := <operation4>
- <operation4> := <operation3> { ‘|’ <operation3> }
- <operation3> := <operation2> { & <operation2>}
- <operation2> := <operation1> { (+ | -) <operation1> }
- <operation1> := <string> { (* | / | %) <string> }
- <string> := ‘(‘<expression>’)’ | <var> | func’(‘<expression>, <expression>’)’ | not’(‘<expression>’)’ | <num>

---

11 Examples

Here are some examples:

Example 1: Input file

```
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)
```

Example 1: Translated LLVM IR file

```
; 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* %d
%16 = load i32, i32* %bb
%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)
%f = alloca i32
store i32 0, i32* %f
%g = alloca i32
%20 = load i32, i32* %f
%21 = urem i32 1,32
%22 = shl i32 %20,%21
%23 = sub i32 32,%21
%24 = ashr i32 %20,%23
%25 = or i32 %22,%24
%26 = load i32, i32* %a
%27 = sub i32 0,%26
%28 = add i32 %27,12
%29 = mul i32 %25,%28
store i32 %29, i32* %g
%h = alloca i32
%30 = load i32, i32* %c
%31 = ashr i32 %30,2
%32 = load i32, i32* %e
%33 = or i32 %31,%32
store i32 %33, i32* %h
%hh = alloca i32
store i32 0, i32* %hh
%34 = load i32, i32* %g
%35 = load i32, i32* %hh
%36 = xor i32 %34,%35
%37 = xor i32 %36, -1
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %37)
%i = alloca i32
%39 = load i32, i32* %g
%40 = urem i32 2,32
%41 = ashr i32 %39,%40
```

```
%42 = sub i32 32,%40
%43 = shl i32 %39,%42
%44 = or i32 %41,%43
%45 = load i32, i32* %g
%46 = load i32, i32* %bb
%47 = xor i32 %45,%46
%48 = xor i32 %47, -1
%49 = add i32 %44,%48
store i32 %49, i32* %i
%50 = load i32, i32* %i
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %50)
%52 = load i32, i32* %h
%53 = add i32 %52,38
%54 = shl i32 %53,1
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %54)
ret i32 0
}
```

Example 1: Output of LLVM IR file

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

Example 1: Errors printed to the terminal

As there are no errors, nothing has been printed into the terminal.

Example 2: Input file

```
a = 5
b = a * 3
a * b - a + b & a * a + b * b - a | b * a - b + a
c = a + b
c * a - b + a & b * c + a * a - b | c * b - a + c
a * c - b + c & a * b + c * c - a | b * a - c + a
```

Example 2: Translated LLVM IR file

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

define i32 @main() {
%a = alloca i32
store i32 5, i32* %a
%b = alloca i32
%1 = load i32, i32* %a
%2 = mul i32 %1,3
store i32 %2, i32* %b
```

```
%3 = load i32, i32* %a
%4 = load i32, i32* %b
%5 = mul i32 %3,%4
%6 = load i32, i32* %a
%7 = sub i32 %5,%6
%8 = load i32, i32* %b
%9 = add i32 %7,%8
%10 = load i32, i32* %a
%11 = load i32, i32* %a
%12 = mul i32 %10,%11
%13 = load i32, i32* %b
%14 = load i32, i32* %b
%15 = mul i32 %13,%14
%16 = add i32 %12,%15
%17 = load i32, i32* %a
%18 = sub i32 %16,%17
%19 = and i32 %9,%18
%20 = load i32, i32* %b
%21 = load i32, i32* %a
%22 = mul i32 %20,%21
%23 = load i32, i32* %b
%24 = sub i32 %22,%23
%25 = load i32, i32* %a
%26 = add i32 %24,%25
%27 = or i32 %19,%26
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %27)
%c = alloca i32
%29 = load i32, i32* %a
%30 = load i32, i32* %b
%31 = add i32 %29,%30
store i32 %31, i32* %c
%32 = load i32, i32* %c
%33 = load i32, i32* %a
%34 = mul i32 %32,%33
%35 = load i32, i32* %b
%36 = sub i32 %34,%35
%37 = load i32, i32* %a
%38 = add i32 %36,%37
%39 = load i32, i32* %b
%40 = load i32, i32* %c
%41 = mul i32 %39,%40
%42 = load i32, i32* %a
%43 = load i32, i32* %a
%44 = mul i32 %42,%43
%45 = add i32 %41,%44
%46 = load i32, i32* %b
%47 = sub i32 %45,%46
%48 = and i32 %38,%47
%49 = load i32, i32* %c
%50 = load i32, i32* %b
%51 = mul i32 %49,%50
```

```
%52 = load i32, i32* %a
%53 = sub i32 %51,%52
%54 = load i32, i32* %c
%55 = add i32 %53,%54
%56 = or i32 %48,%55
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %56)
%58 = load i32, i32* %a
%59 = load i32, i32* %c
%60 = mul i32 %58,%59
%61 = load i32, i32* %b
%62 = sub i32 %60,%61
%63 = load i32, i32* %c
%64 = add i32 %62,%63
%65 = load i32, i32* %a
%66 = load i32, i32* %b
%67 = mul i32 %65,%66
%68 = load i32, i32* %c
%69 = load i32, i32* %c
%70 = mul i32 %68,%69
%71 = add i32 %67,%70
%72 = load i32, i32* %a
%73 = sub i32 %71,%72
%74 = and i32 %64,%73
%75 = load i32, i32* %b
%76 = load i32, i32* %a
%77 = mul i32 %75,%76
%78 = load i32, i32* %c
%79 = sub i32 %77,%78
%80 = load i32, i32* %a
%81 = add i32 %79,%80
%82 = or i32 %74,%81
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %82)
ret i32 0
}
```

Example 2: Output of LLVM IR file

```
85
315
124
```

Example 2: Errors printed to the terminal

As there are no errors, nothing has been printed into the terminal.

Example 3: Input file

```
Kuzurete = 99
Yuku = 75
Maeni = 12
Kuzurete + Yuku | 1
Maeni * Kuzurete | 1
Yuku - Maeni | 1
```

- Kuzurete
& Yuku & 1
xor(Maeni &)1
xor(Kuzurete * Yuku * maeni, 0 % 0)
xor(Kuzurete * Yuku * maeni, 0) % 0
xor(Kuzurete * Yuku * Maeni, 0

Example 3: Translated LLVM IR file

As there are errors inside the input file, no output file has been created.

Example 3: Output of LLVM IR file

As there are errors inside the input file, no output file has been created.

Example 3: Errors printed to the terminal

Error on line 7!
Error on line 8!
Error on line 9!
Error on line 10!
Error on line 11!
Error on line 12!