Bogazici University

CmpE 230: System Programming

Spring 2023

Project 2 – Transcompiler

Ahmet Ertuğrul Hacıoğlu

2020400132

Ceyhun Sonyürek

2020400258

Introduction

In this project, a transpiler which translates statements and expressions of the AdvCalc++ language into LLVM IR code is implemented in C programming language.

Purpose

The purpose of the project is generating a low-level code calculator that can compute complicated expressions which may include many functions and nested operations and store many variables assigned to use them in calculations.

Design

Firstly, the code reads an input file which has mathematical expressions on each line and parses them to lexemes which are smallest meaningful pieces line by line. After that, the code adds these tokens(lexemes) into a double linkedlist. Then, an error detector part starts iterating linkedlist. This part checks every token with their previous and next nodes (holders of the tokens in linkedlist) whether they can appear in order or not. If the form of the input is legal, code reaches the last part of the main function. In the last part, the code checks if there is an assignment operation. If so, the code stores the value of variable after expression performs. Also the code allocates memory for a variable before calculations when a variable name occurs first time. After doing this, the code assign register names to all variables in expression and calls a function for remaining complex calculations. In these function, starting from head, it iterates linkedlist with respect to operation or function precedence. Starting from uppermost function or paranthesis, the code prints LLVM IR code into a output file and replaces the register name with current data of the node and delete other nodes which is part of function or operations. In doing so, the code continues the removing other nodes and replacing its register name into the node until only one node remains.

Implementation

- Data Types

A *Node* struct which has 7 members to store every token in a linkedlist format for every lexeme in inputs. “data” member stores the smallest token which has a meaning in lexemes or if the *Node* is a variable it “data” holds its register name and “variableName” holds name of the variable. ”type” member of *Node* is 0 if it is a number and 1 if it is a register converted from a variable. “next” and “prev” members helps accessing elements in appropriate order. “func” and “var” members are used for checking the token contains alphabetic character if it is a function or a variable.

Global variables of *fpWrite and *test holds input and output names and counter is the register number used in program. The code also has many local variables with every type in the scope of functions and main.

- Functions

addLast: Adds the Node that has taen as a parameter to the end of the double linkedlist by iterating. The function also checks if the token is a function name or variable or neither of them.

operations: In operations function, it is assumed that there is no function and parenthesis.If the temp node(starting from head and traverse linked list one by one) encounter with parenthesis or function code break.Since there is no function, if temp node’s data “*” or other mathematical operation, the code prints the data of previous node and following node.Data of prev or next node is registerID or number since there is no variable(main function convert variable into register by changing node’s data to registerID which is loading itself).After printing it, the code delete the temp and next node and changing the data of previous node of temp to new registerID. Also, the code should change the previous node’s type since after the operation, instead of numbers only registers remain. When the code traverse linked list once, it should return to head and check other operations according to operator precedence in C language.

paren: In paren function, we need to find the number of paranthesis in order to find the innermost parenthesis.When we find innermost parenthesis, we need to call operations function since inside the innermost parenthesis there is no variable, function or another parenthesis.Then there is three node left in the part we consider in linked list which are “(“,temp node and “)”. The code remove the parenthesis by changing first parenthesis data to temp node’s data and delete the other node and we reduce the number of element in linkedlist. When there is no left parenthesis we are done with this function.

inside_func: In this function, we need to find innermost function and call paren(inside paren of course operations) in order to there is just function name and its parameters.Starting with outermost function, we need to find the number of functions before comma to find innermost function by counting paranthesis. Flag is the boolean that determine the last node we should traverse.If flag is true, this means we need to break before comma, otherwise we need to break closed bracket of outermost function.After this steps, we need to print the function and its parameter and change the type of node to register.

terminate: Terminate function traverses all linked list and call other functions. If we encounter with function, the code calls inside function twice in order to delete nodes and remains just outermost function and its parameters.When there is no left function, it calls paren function in order to do operations and remove paranthesis and return just one node which should store to allocated variable.

main: At the beginning of the main function, input file and output file are instantiated. First lines of LLVM IR code is written in output file. Then, a while loop starts reading input file line by line and increments linechecker variable for every iteration of a new line. Each line of input is read with an for loop that parses characters by their type such as alphabetic or numeric. After loop get smallest meaningful token, it calls addLast function to add them into linkedlist.

```
if (input[i] == '/') {          // An if-block example for parsing
    char token[] = "/";
    addLast(&head, token);
    continue;
}
```

After the end of the for-loop, error detection parts starts with checking number of parenthesis are valid. Then, a while loop starts to iterate over linkedlist of tokens to check if the order of lexemes are legal.

```
if (isalpha((tkn->prev->data)[0]) && isdigit((tkn->data)[0])) {
    printf("Error on line %d!\n", linechecker); // An example if-block for error detecting
    isThereError = true;
    globError = true;
    break;
}
```

After the error detector part, the codes checks whether the line is an assignment statement or not. If it is an assignment statement, it prints an storing statement for variable value. Then, it also checks if the variable before equal sign is initialized and if not, it prints a allocation statement in LLVM IR code in output file. The code also converts all variable to a register in this part and calls terminate function to translate complex operations to LLVM IR form.

```
strcpy(tok->variableName, tok->data);
char b[256 + 1];     // An example of register loading
sprintf(b, "%%rg%d", counter);
strcpy(tok->data, b);
fprintf(fpWrite, "%s = load i32, i32* %%%s\n", tok->data, tok->variableName);
counter++;
tok->type = 1;
```

In the last part of the main function code terminates the output file if there is occured an error before. If not, it succesfully completes the code translation.

Using The Code

• Type make in the command line and then type ./advcalc to initialize program.

• This program supports the following operations and functions: +, -, *,%, &, |, xor, ls, rs, lr , rr,

not.

• Variable use is also supported, once a variable is initialized, it can be used throughout the

program. <var> = <expression> syntax is used to assign a value to a variable.

• Every value and every calculation should be 32-bit integer-valued.

• The language does not support the unary minus (-) operator (i.e x = -5 or a = -b is not valid).

However, subtraction operation is allowed.

• The variable names should consist of lowercase and uppercase Latin characters in the

English alphabet [a-zA-Z].

• Expressions or assignments should consist of 256 characters at most.

Input & Output Examples

Input

```
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)
```

Output

```
; 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

%rg1 = load i32, i32* %a

%rg2 = mul i32 %rg1,5

store i32 %rg2, i32* %bb
```

```
%c = alloca i32
%rg3 = load i32, i32* %a
%rg4 = load i32, i32* %bb
%rg5 = sub i32 %rg3,2
%rg6 = mul i32 %rg5,%rg4
store i32 %rg6, i32* %c
%d = alloca i32
%rg7 = load i32, i32* %a
%rg8 = shl i32 %rg7,3
store i32 %rg8, i32* %d
%rg9 = load i32, i32* %c
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %rg9)
%rg10 = load i32, i32* %d
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %rg10)
%rg11 = load i32, i32* %bb
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %rg11)
%e = alloca i32
%rg12 = load i32, i32* %d
%rg13 = load i32, i32* %bb
%rg14 = xor i32 %rg12,%rg13
store i32 %rg14, i32* %e
%rg15 = load i32, i32* %e
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %rg15)
%f = alloca i32
store i32 0, i32* %f
%g = alloca i32
%rg16 = load i32, i32* %f
%rg17 = load i32, i32* %a
%rg18 = shl i32 %rg16,1
%rg19 = sub i32 32,1
%rg20 = lshr i32 %rg16,%rg19
%rg21 = or i32 %rg18,%rg20
%rg22 = sub i32 0,%rg17
%rg23 = add i32 %rg22,12
%rg24 = mul i32 %rg21,%rg23
store i32 %rg24, i32* %g
%h = alloca i32
%rg25 = load i32, i32* %c
```

```
%rg26 = load i32, i32* %e
%rg27 = ashr i32 %rg25,2
%rg28 = or i32 %rg27,%rg26
store i32 %rg28, i32* %h
%hh = alloca i32
store i32 0, i32* %hh
%rg29 = load i32, i32* %g
%rg30 = load i32, i32* %hh
%rg31 = xor i32 %rg29,%rg30
%rg32 = xor i32-1 ,%rg31
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %rg32)
%i = alloca i32
%rg33 = load i32, i32* %g
%rg34 = load i32, i32* %g
%rg35 = load i32, i32* %bb
%rg36 = lshr i32 %rg33,2
%rg37 = sub i32 32,2
%rg38 = shl i32 %rg33,%rg37
%rg39 = or i32 %rg36,%rg38
%rg40 = xor i32 %rg34,%rg35
%rg41 = xor i32-1 ,%rg40
%rg42 = add i32 %rg39,%rg41
store i32 %rg42, i32* %i
%rg43 = load i32, i32* %i
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %rg43)
%rg44 = load i32, i32* %h
%rg45 = add i32 %rg44,38
%rg46 = shl i32 %rg45,1
call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %rg46)
ret i32 0
}
```

Difficulties Encountered

Most important difficulty or problem I encounter that while removing the nodes and change the data of temp node, I forgot the change of the type of node.For example;

x = 3;

y = 2 + 5 * x;

First we allocate x and store.After that, in second line we change the type of x to register and change the data of x into rg"number".The main reason for changing the type of variable into register is that when we do operations we print the data whether the temp's type is register or number.

When we multiply 5 and rg%1, we remove "*" and rg%1 nodes and change the data of 5 to rg%2 and if we forgot the type of 5 (number) to register, we can encounter problems.This problem can occur if there is a function.If we don't change the function bool,we can suppose it is still function while it is converted into register.

Improvements and Extensions

We always traverse iteratively and count the number of functions or paranthesis.I think we can write the functions with using recursion.