Bogazici University

TRANSCOMPILER DOCUMENTATION

Ahmet Fırat Gamsız
2020400180
Department of Computer Engineering

Ali Tarık Şahin
2020400207
Department of Computer Engineering

CmpE 230 Systems Programming
Prof Dr Can Özturan
May 1, 2023

Introduction

ADVCALC2IR is a transpiler written in C programming language that translates AdvCalc++ language into LLVM IR code. ADVCALC2IR accepts lines of expressions and assignment statements as input and outputs LLVM IR correspondence.

Purpose

This project aims to implement a fully functional version of ADVCALC2IR. Transcompiler must be able to translate input code of AdvCalc++ into LLVM IR code. AdvCalc++ is able to evaluate the given set of operations and functions, assign return values to variables and retrieve stored values from variables. LLVM IR is a low-level intermediate representation and uses static single assignment-based representation. The transcompiler must be able to detect and log errors.

Design

The design of the calculator consists of three parts: Analyzer, Computer, Main.

Analyzer is the part where user input is turned into a computable form. Analyzer consists of a lexer, syntax checker and reformatter. Since design principles of this part remain the same as AdvCalc we refrain from duplicating same information and recommend resorting to AdvCalc documentation.

Computer is the part where all the calculations are made with respect to precedence. It takes the head of the token list and iterates through it as many as the number of precedence levels. Starting from the most significant operation or function, it does the simulation of computation and writes corresponding instructions into file.ll. As doing so, it gets token list shrunk. At the end, the remaining register in the token list will have the resulting value, which, in turn, can be used while printing or assigning. Mostly, this algorithm is a loop-based algorithm, but the only exception is parentheses cases. Considering expressions inside parentheses as a distinct problem leads the algorithm to use recursion to handle parentheses.

Main is the conductor of the program. Reading input, initializing necessary variables, printing to the output file and handling errors is provided by Main. Also, Main performs as a data pipeline for Analyzer and Computer. Functions' inputs and outputs are connected via Main. Unlike AdvCalc, Main reads input from file and prints its output into another file. Errors do not terminate the program, but the output file gets deleted, and errors are logged in terminal.

Implementation

Data types

This project uses various data types for different tasks. An enumerated type called token_type is created to identify different types of tokens. A full list of token types and corresponding definitions are given in the Appendix. A struct called token is defined to carry lexemes and their related information. These tokens are connected to each other to form a doubly linked list. Token struct holds type of the token as token_type, token value as string, related register's name as string and pointer to next and previous token. Reserved keywords are held in globally initialized string array. Valid signs are held in the same way but as character array. For global variables that AdvCalc stores a lookup table has been used. This lookup table consists of a string array for variable keys, long long array of actual values and integer index that points to the next empty position in lookup table. String positions in the array correspond to the actual values and access is provided by this correspondence. The index is manually kept up to date each time a new variable is defined. Lookup tables elements are global variables. Two extra global indexes are included as register index and line index. These indexes are also manually updated and are not regulated. Line index is kept for correctly reporting error locations and register index holds the next free LLVM register index.

Functions

Lexer: This function converts user's string input to a linked list of tokens. Its main principle is looping over every char by itself or its subfunctions and extracting tokens from every lexeme. For further information one may refer to AdvCalc documentation. Only important difference in AdvCalc++ implementation is that comment token was removed. Therefore, "%" is not converted into EOL token but modulo token.

Syntax checker: This function checks the correctness of syntax of given linked list. If an equal sign is found, the linked list is scanned until the sign's pointer to check the rules for left hand side. Then the right-hand side (or full expression if no equal sign was found) is passed to expression checker function. This function iterates every token to check the whole expression. Each token is checked by a list of predefined rules. For further information one may refer to AdvCalc documentation.

Reformatter: This function replaces commas with their corresponding functions and replaces variables with their actual values. For further information one may refer to AdvCalc documentation. When a variable is seen lookup table is queried and if a value is found token is converted to an integer token with that value. Also, a register name assigned to this token and load command is called in LLVM. The query returns error if a variable was not found.

Calculator: Calculate is a void type function whose aim is to write steps and resulting values into file.ll. It loops through the given input for every level of precedence and handles operations one at a time, from higher precedence to lower precedence. (e.g. It will handle multiplication before additon) Here is some code snippet where one of those succesive loops are implemented:

```
//Loop for addition and subtraction

temp_head = head;


while (temp_head->token_type != CLOSE_P && temp_head->token_type != EOL) {

    if (temp_head->token_type == SUM) {

        calculate_opr(temp_head, SUM);

    }

    if (temp_head->token_type == MINUS) {
        calculate_opr(temp_head, MINUS);
    }
    temp_head = temp_head->next;
}
```

This calculate function is also responsible for handling parenthesis situation. It recursively calls itself whenever an opening parenthesis is seen and ends the recursive level whenever a closing parenthesis is seen. This property also prioritize the expressions with parenthesis over other expressions.

```
//Loop for parentheses
    while (temp_head->token_type != CLOSE_P && temp_head->token_type != EOL) {
        if (temp_head->token_type == OPEN_P) {
            calculate(temp_head->next);
        }
        temp_head = temp_head->next;
    }
```

On the other hand, Calculate function uses a helper function called Calculate_opr which is another void type function. Calculate_opr handles all the binary operations and since binary functions like xor, ls, rr are turned into binary operations at early stages of the program (e.g. xor(a, b) is, at this stage, (a xor b)) they can also be treated as binary operations. Calculate_opr takes a pointer to operator and takes its type. It uses a switch case structure to find corresponding operator and write the script accordingly. It also keeps track of a counter called REG_IDX, which is used to handle register naming. Here is a code snippet of one of the switch cases:

```
        case SUM:
            sprintf(new_register_name, "%%reg%d", REG_IDX);
            REG_IDX++;
            fprintf(op,"\t%s = add i32 %s, %s\n", new_register_name, left_register_name,
            right_register_name);
            break;
```

It assigns the left register and the right register into their corresponding places and outputs new_register_name. At the end of this function it assigns this new_register_name to char register_name[16] inside token struct, which probably be used as a left register or right register later on.

How to Use

• Type *make* in the command line and that’s it.

• 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.

• Undefined variables will result in error.

Input Output Example

Input:

```
1. x = 1
2. y = x + 3
3. z = x * y * y*y
4. z
5. xor(((x)), x)
6. xor(((x)), x) | z + y
7. rs(xor(((x)), x) | z + y, 1)
8. ls(rs(xor(((x)), x) | z + y, 1), (((1))))
```

Output:

```
; ModuleID = 'advcalc2ir'

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

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


define i32 @main() {

        %x = alloca i32

        store i32 1, i32* %x

        %reg1 = load i32, i32* %x

        %reg2 = add i32 %reg1, 3

        %y = alloca i32

        store i32 %reg2, i32* %y

        %reg3 = load i32, i32* %x

        %reg4 = load i32, i32* %y

        %reg5 = load i32, i32* %y

        %reg6 = load i32, i32* %y
        %reg7 = mul i32 %reg3, %reg4

        %reg8 = mul i32 %reg7, %reg5

        %reg9 = mul i32 %reg8, %reg6

        %z = alloca i32

        store i32 %reg9, i32* %z

        %reg10 = load i32, i32* %z

        call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str,
i32 0, i32 0), i32 %reg10)

        %reg11 = load i32, i32* %x

        %reg12 = load i32, i32* %x

        %reg13 = xor i32 %reg11, %reg12

        call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str,
i32 0, i32 0), i32 %reg13)

        %reg14 = load i32, i32* %x

        %reg15 = load i32, i32* %x

        %reg16 = load i32, i32* %z

        %reg17 = load i32, i32* %y

        %reg18 = xor i32 %reg14, %reg15

        %reg19 = add i32 %reg16, %reg17

        %reg20 = or i32 %reg18, %reg19

        call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str,
i32 0, i32 0), i32 %reg20)

        %reg21 = load i32, i32* %x

        %reg22 = load i32, i32* %x

        %reg23 = load i32, i32* %z

        %reg24 = load i32, i32* %y

        %reg25 = xor i32 %reg21, %reg22

        %reg26 = add i32 %reg23, %reg24

        %reg27 = or i32 %reg25, %reg26

        %reg28 = ashr i32 %reg27, 1

        call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str,
i32 0, i32 0), i32 %reg28)

        %reg29 = load i32, i32* %x

        %reg30 = load i32, i32* %x

        %reg31 = load i32, i32* %z

        %reg32 = load i32, i32* %y

        %reg33 = xor i32 %reg29, %reg30
        %reg34 = add i32 %reg31, %reg32

        %reg35 = or i32 %reg33, %reg34

        %reg36 = ashr i32 %reg35, 1

        %reg37 = shl i32 %reg36, 1

        call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str,
i32 0, i32 0), i32 %reg37)


        ret i32 0

}
```

Difficulties Encountered

- Different from advcalc project, calculate function needs to be void type, hence it required us to work with pointers more when dealing with recursive calls, since this time there is no returned value.
- Rotation operations in calculate_opr have no peculiar keywords in llvm, so they have to be implemented as a mixture of other operations, which resulted in several lines of instructions and several registers.

Discussion

Why did we use a bottom-up approach?

This approach helped us to write all instructions in order, from the highest precedence to the lowest. Thanks to this method, we were able to turn advcalc project into advcalc++ with only a few lines of print(fprintf) statements at places where we do operations.

Why did we use linked list implementation?

When turning calculate function's return type into void, some problems regarding return values raised. We were able to access returned values(registers) through pointer syntax with the help of linked list implementation.

What is the complexity of the program?

All the functionality of the program can be reduced to O(N) time complexity. Functional costs are just a matter of a couple of loops through the input size, which is roughly 8-9 loops. Considering O(9*N) complexity, it can be regarded as linear time complexity.

Conclusion

Using the advantages of the C programming language, we were able to produce a fast, easy to use transpiler and a maintainable, easy to read code repository behind it.

Appendix

Token types and their definitions

`VAR` -> Variable

`INT` -> Integer

`OPEN_P` -> Open parenthesis

`CLOSE_P` -> Closed parenthesis

`SUM` -> + sign

`MULTI` -> * sign

`DIV` -> / sign

`MOD` -> % sign

`MINUS` -> - sign

`EQUAL` -> = sign

`B_AND` -> & sign

`B_OR` -> | sign

`B_XOR` -> xor function

`LS` -> ls function

`RS` -> rs function

`LR` -> lr function

`RR` -> rr function

`NOT` -> not function

`COMMA` -> , sign

`EOL` -> End of line