Project 2 - Transcompiler

CmpE 230, Systems Programming, Spring 2023

Instructor: Can Özturan
TA: Gökçe Uludoğan
SAs: Bahadır Gezer, Ömer Talip Akalın

Due: 26/04/2023, 23:55 Sharp

1 Introduction

Welcome to yet another C programming project. In this project, you will develop 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.

A transpiler, also known as a source-to-source compiler, is a type of compiler that translates source code from one programming language to another. In this case, our program will translate the input code into LLVM IR code, which is a low-level intermediate representation that can be used to generate code for various target architectures.

2 Details

In this project, you will develop a translator called ADVCALC2IR that will take input in the form of assignment statements and expressions -one on each line- and generate low-level LLVM IR code that can compute and output these statements. The LLVM IR code generated will be in the form of static single assignment (SSA) based representation, where variables are assigned a value once.

The LLVM IR code generated by the ADVCALC2IR translator will use the `alloca` keyword to allocate space for variables and return the address of allocation. Variables in the LLVM IR code will start with the `'%'` sign. The project will only use 8-bit, 16-bit, and 32-bit integer operations, and you can assume only binary operations.

The generated IR code will include a module name and a prototype for the `printf` output statement, which will be used to print the value of a variable using the `printf` function. Example code can be used directly in translations.

Errors should be reported if there is a syntax error or if an undefined variable is used. Additionally, all errors should be reported together with the line numbers. The specific format for error reporting is detailed below.

You can assume that only binary operations will be used in the expressions. The project provides a set of operations that can be used in the expressions, including addition, multiplication, subtraction, division, bitwise operations, modulus, and several bitwise shift and rotation operations. All variables and operations are integer operations.

Use the following commands to compile, run, and test the generated code:

Example Command	Explanation
`./advcalc2ir file.adv`	Runs ADVCALC2IR on `file.adv` and produces IR code in `file.ll`.
`lli file.ll`	Runs the LLVM interpreter and dynamic compiler. This command produces the output based on the given example code.
`llc file.ll -o file.s`	Invokes the `llc` compiler to produce the assembly code.
`clang file.s -o myexec`	Compiles the assembly code to produce the executable.
`./myexec`	Runs the executable.

As a summary the information above note the following about the IR code:

- 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.
- Integer variable names denote temporary variables.
- 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 )
```

  The IR line above prints the value of variable `%7`. Lines 20 and 37 on example 1, 15 and 29 on example 2 use this function to print variable values. Also the line of code above is a single line, the '\' breaks the line into two lines for aesthetic purposes.

The input .adv files will use a language that has the following properties:

- 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.
- '`%`' characters denote comments. Any characters after '`%`' will be considered as a comment, not code.
- You must run and test your code on an Ubuntu Linux machine before submitting. You can use a Linux virtual machine or WSL in this context.
- The input `advcalc++` language may include all sorts of syntax errors.
- Unlike `advcalc`, 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)

Operation	Explanation
a + b	Returns summation of a and b.
a * b	Returns multiplication of a and b.
a - b	Returns the subtraction of b from a. No unary minus.
a / b	Returns the quotient of the division.
a & b	Returns bitwise a and b.
a | b	Returns bitwise a or b.
a % b	Returns a modulo of b.
xor(a, b)	Returns bitwise a xor b.
ls(a, i)	Returns the result of a shifted i bits to the left.
rs(a, i)	Returns the result of a shifted i bits to the right.
lr(a, i)	Returns the result of a rotated i times to the left.
rr(a, i)	Returns the result of a rotated i times to the right.
not(a)	Returns bitwise complement of a.

3 Submission

Your project will be tested automatically. Thus, it's important that you carefully follow the submission instructions. The root folder for the project should be named according to your student id numbers. If you submit individually, name your root folder with your student id. If you submit as a group of two, name your root folder with both student ids separated by an underscore. You will compress this root folder and submit it with the `.zip` file format. Other archive formats will not be accepted. The final submission file should be in the form of either `2020400039.zip` or `2019400046_2020400039.zip`. Submissions will be done through Moodle.

You must create a Makefile in your root folder which creates a `advcalc2ir` executable in the root folder, do not include this executable in your submission, it will be generated with the `make` command using the Makefile. The `make` command should not run the executable, it should only compile your program and create the executable.

Late Submission

If the project is submitted late, the following penalties will be applied:

Hours Late	Penalty
$0 <$ hours late ≤ 24	25%
$24 <$ hours late ≤ 48	50%
$48 <$ hours late	100%

4 Grading

Your project will be graded according to the following criteria:

- **Code Comments (8%):** Write code comments for discrete code behavior and method comments. This sub-grading evaluates the quality and quantity of comments included in the code. Comments are essential for understanding the code's purpose, structure, logic, and any complex algorithms or data structures used. The code should be easily readable and maintainable for future developers.
- **Documentation (12%):** A written document describing how you implemented your project. This sub-grading assesses the quality and completeness of the written documentation accompanying the project. Good documentation should describe the purpose, design, and implementation details of the project, as well as any challenges encountered and how they were addressed. The documentation should also include examples of input/output and how to use the program. Students should aim to write clear, concise, and well-organized documentation that effectively communicates the project's functionality and design decisions.
- **Implementation and Tests (80%):** The submitted project should be implemented following the guidelines in this description and should pass testing. This sub-grading assesses the quality and correctness of the implemented project, including its functionality and accuracy in handling expressions and assignment statements.

5 Warnings

- You can submit this project either individually or as a group of two.
- All source codes are checked automatically for similarity with other submissions and exercises from previous years. Make sure you write and submit your own code. Any sign of cheating will be penalized with an F grade.
- Do not use content from external AI tools directly in your code and documentation. Doing so will be viewed as plagiarism and thoroughly investigated.
- Project documentation should be structured in a proper manner. The documentation must be submitted as a .pdf file. Also include a .txt file which has the same contents as the .pdf file. This text file should have the same file name as the pdf, and the same content, just a different file extension.
- Make sure you document your code with necessary inline comments and use meaningful variable names. Do not over-comment, or make your variable names unnecessarily long. This is very important for partial grading.
- Do not start coding right away. Think about the structure of your program and the possible complications resulting from it before coding.
- Questions about the project should be sent through the discussion forum on Piazza.
- Only use the C programming language for the transpiler source code.
- Your programs will be tested in a Linux Ubuntu environment.
- There will be a time limit of 30 seconds for your `advcalc2ir` program execution. Program execution consists of the total amount of time for all queries. This is not a strict time limit and the execution times may vary for each run. Thus, objections regarding time limits will be considered if necessary.

Figure 1: Random Cat Picture

6 Examples

Below are some examples that may be useful.

Example 1:

`advcalc++`:

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

Translated file in IR form:

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

define i32 @main() {
    %k = alloca i32
    %x = alloca i32
    %y = alloca i32
    %zvalue = alloca i32
    store i32 3, i32* %x
    store i32 5, i32* %y
    %1 = load i32, i32* %x
    %2 = load i32, i32* %y
    %3 = add i32 1,%2
    %4 = mul i32 %1,%3
    %5 = add i32 23,%4
    store i32 %5, i32* %zvalue
    %6 = load i32, i32* %zvalue
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %6 )
    %8 = load i32, i32* %x
    %9 = load i32, i32* %y
    %10 = sub i32 %8,%9
    %11 = load i32, i32* %zvalue
    %12 = sub i32 %10,%11
    store i32 %12, i32* %k
    %13 = load i32, i32* %x
    %14 = load i32, i32* %y
    %15 = mul i32 3,%14
    %16 = add i32 2,5
    %17 = mul i32 1,%16
    %18 = mul i32 %15,%17
    %19 = add i32 %13,%18
    store i32 %19, i32* %k
    %20 = load i32, i32* %k
    %21 = add i32 %20,1
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %21 )
    ret i32 0
}
```

Both executing the `lli` command with the IR file and running the executable generated by using `llc` and `clang` commands should give the following output:

```
41
109
```

Example 2:

`advcalc++`:

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

Translated file in IR form:

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

define i32 @main() {
    %siuuu = alloca i32
    %siu = alloca i32
    store i32 0, i32* %siuuu
    store i32 0, i32* %siu
    store i32 11, i32* %siu
    store i32 7, i32* %siuuu
    %sad1 = load i32, i32* %siu
    %sad3 = load i32, i32* %siuuu
    %sad2 = sdiv i32 %sad1, %sad3
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %sad2 )
    %sad4 = load i32, i32* %siu
    %sad6 = load i32, i32* %siuuu
    %sad5 = mul i32 %sad4, %sad6
    store i32 %sad5, i32* %siu
    %sad7 = load i32, i32* %siu
    %sad9 = load i32, i32* %siu
    %sad10 = sub i32 %sad7, %sad9
    %sad11 = load i32, i32* %siu
    %sad13 = load i32, i32* %siu
    %sad14 = mul i32 %sad11, %sad13
    %sad15 = load i32, i32* %siu
    %sad12 = sdiv i32 %sad14, %sad15
    %sad8 = add i32 %sad10, %sad12
    call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0, i32 0), i32 %sad8 )

    ret i32 0
}
```

Executing `lli file.ll` or `./file.exe` should give the following:

```
1
77
```