**CS544 Project Proposal - LLVM to IntCode Compiler**

Erich Schwarzrock

For my project I wanted to take the WPL compiler one step further and create a compiler for the LLVM code which compiles it into a fake machine language. The “machine” language I was planning to use is called “IntCode”. IntCode comes from the 2019 Advent of Code competition, which is a yearly Christmas themed set of small coding challenges in the style of an advent calendar. This language only has 11 commands, add, multiply, input, output, less than, equal to, jump if true, jump if false, adjust stack pointer, terminate program, and divide (which I added to the spec because I think it would be too hard to implement division with just the add and multiply instructions). I wrote out the code spec and is at the bottom of this report.

I chose IntCode over real machine languages because it is much simpler, I already have prior experience with it, and there isn’t already a compiler out there for it. I believe that IntCode is still restrictive enough to be a large enough challenge for this project, since it only has those 11 basic commands and no variables. Also, some of the things I think I will have to do within IntCode include, creating and managing a stack so I can support recursive programs, load data from the stack when needed, create a function to print strings, and generally figure out how to do logical operations only using add, multiply, less than, and equal to. These tasks I believe will give me a basic understanding of how machine languages handle these tasks, so that this project teaches me something useful, and will provide enough of a challenge to be a substantial project.

As far as the compiler goes, I will try to support the entire WPL language, as I already have a working WPL to LLVM compiler, however I will not be able to support C standard functions. Instead I plan to create 4 of my own external functions which will be input an int, print an integer as int(printInt(65) prints “65”, print an intger as ascii char(printIntAsChar(65) prints “A”), and print string.

As far as implementation goes, I believe I will have to create a lexer, symbol table for variables and functions, code visitor, and within IntCode I will have to create a stack structure and the 4 external functions I mentioned above.

I chose to do this project because I competed in the Advent of Code back in 2019, thanks to Prof. Heineman, which had us make this machine little by little. Once we completed the full spec on day 9 we continued to use the machine and the competition would give us programs to run, and most of these programs were little games. This got me wondering how the event organizer was able to create all those games with just IntCode, and that got me wanting to create my own compiler for IntCode, which lead me to this class, 3 years later. Once we started working with LLVM I saw how basic the instructions in LLVM were and that it wouldn’t be too hard to translate them into IntCode, it would just take a little problem solving and figuring out how to implement a stack. So basically, I’ve wanted to do this for a while and now I can. Also I already have a working IntCode machine that I can use from when I first did the competition.

For this project I intend to use my WPL compiler to get the LLVM code, and I will build the LLVM to IntCode compiler in Java, using no compiler specific libraries. I am choosing Java because I am on windows and getting c, c++, c# to run is always a hassle and I already have java running on my machine. Also, I am much more familiar with Java. In addition, I don’t think the compiler will be too complex, because I did some preliminary research and found I only use around 35 unique LLVM commands to compile WPL. This means I will only have 35 unique LLVM commands to compile and I think I can forgo creating a full antlr grammar. Also, since LLVM is a very linear language, and all the code is error free (since I did error checking in my original WPL compiler) I won’t need to create complex structures to compile it and do complex error checking. I think all I will need is a symbol table for variables and a symbol table for functions/blocks, specifically storing where each block starts.

**Int Code Specs**

An Intcode program is a list of integers separated by commas (like 1,0,0,3,99). To run one, start by looking at the first integer (called position 0). Here, you will find an opcode - either 1, 2, or 99. The opcode indicates what to do. Once you're done processing an opcode, move to the next one by stepping forward 1 + numAgs positions.

(For syntax leading 0s can be dropped)

01 - add

Sytax - 01, &a, &b, &c

Operation in C - c = a + b;

adds together numbers read from two positions and stores the result in a third position. The three integers immediately after the opcode tell you these three positions - the first two indicate the positions from which you should read the input values, and the third indicates the position at which the output should be stored.

02 - multiply

Syntax - 02, &a, &b, &c

Operation in C - c = a\*b;

works exactly like opcode 1, except it multiplies the two inputs instead of adding them

03 - input

Syntax - 03, &a

Operation in C - scanf("%d", &a);

takes a single integer as input and saves it to the position given by its only parameter. For example, the instruction 3,50 would take an input value and store it at address 50.

04 - output

Syntax - 04, &a

Operation in C - printf((char)a);

Opcode 4 outputs the value of its only parameter. For example, the instruction 4,50 would output the value at address 50.

05 - jump-if-true

syntax - 05, &a, &b

Operation in C - if(a != 0){instructionPtr = b;}else{instructionPtr += 3;}

Opcode 5 is jump-if-true: if the first parameter is non-zero, it sets the instruction pointer to the value from the second parameter. Otherwise, it does nothing.

06 - jump-if-false

syntax - 06, &a, &b

Operation in C - if(a == 0){instructionPtr = b;}else{instructionPtr += 3;}

Opcode 6 is jump-if-false: if the first parameter is zero, it sets the instruction pointer to the value from the second parameter. Otherwise, it does nothing.

07 - less than

syntax - 07, &a, &b, &c

Operation in C - if(a < b){c = 1;}else{c = 0;}

Opcode 7 is less than: if the first parameter is less than the second parameter, it stores 1 in the position given by the third parameter. Otherwise, it stores 0.

08 - equal to

syntax - 08, &a, &b, &c

Operation in C - if(a = b){c = 1;}else{c = 0;}

Opcode 8 is equals: if the first parameter is equal to the second parameter, it stores 1 in the position given by the third parameter. Otherwise, it stores 0.

09 - add to relative base

Syntax - 09 &a

Operation in C - relative\_base += a

adjusts the relative base by the value of its only parameter. The relative base increases (or decreases, if the value is negative) by the value of the parameter.

99 - end program

Syntax - 99

//My added opcodes to make my life a little easier

12 - divide

Syntax - 12, &a, &b, &c

Operation in C - c = a/b;

it divides the two inputs using int division, because I don't know how to implement division with just add and multiply commands

The opcodes also support type changing of parameters, like so. An opcode is actually in the form of

ABCDE where DE is the code(01,02,03, etc.) and ABC is the type of the 3rd, 2nd and 1st parameter,

respectively. 0 is of type pointer, 1 is of type int(so use the value it's self), and 2 is of type pointer

but it uses the relative pointer.

For example the code (00102, 6, 3, 5, 99, 2) will set address 5(which is currently 2) to 6 \* (value at address 3) and then exit. Because the opcode 00102 set the first parameter to be of type int because of the 1 in position 3, so we use 6 instead of the value in address 6, and the other 2 parameters are of type pointer.

Example Syntax to make this more clear

say we have opcode 55 which outputs the square of the first perameter.

|  |  |  |
| --- | --- | --- |
| IntCode | Syntax in C | Type of Parameter |
| 055, a | Printf((char) ((\*a)\*(\*a))); | pointer |
| 155, a | Printf((char) ((a)\*(a))); | int |
| 255, a | Printf((char) ((\*(a + relative\_base))\*(\*(a + relative\_base)))); | Pointer, but relative to relative\_base |

where relative\_base is a global variable which starts at 0 but can be modified by opcode 9

\*Leading 0s can be dropped from the opcode int, and if there isn't 3 parameters, so 00002 = 2

Intcode also supports virtual memory, so if we try to set a value to an address that is out of range it will still work, exp (11001, 6, 4, 20, 99) we will still put 6+4 in address 20, even though the code only goes up to address 5.

For the original documentation visit <https://adventofcode.com/2019> days 2, 5, 7, 9. Although some of the documentation is locked in part 2 of the problem, so if you need the full original documentation I can copy all of it over, it’s just very long and wordy.