Alperen Bırçak 2018400006
Çisel Zümbül 2018400093

The given project was to write a compiler that translates a specified language into a lower-level language called "llvm". Our approach was to initially start writing a small part of the project, for which we chose the assignment of variables, to analyze the practical nature of the problem. We decided to determine some reoccurring problems we might encounter, build some tools for them, then build upon that the code which does the actual translating. Biggest challenge we faced was to determine how these two parts of the code should communicate.

First of those tools we wrote was the struct object "lineReader", which is used for tokenizing the input we are given. This object holds a string and lets you interact with it through a stream-like interface, which it accomplishes by using a cursor that points to the relevant parts of the string. It has three public functions for the interaction: "get()" function returns the current alphanumeric word or a symbol then advances the cursor forward, "peek()" function does the same thing without advancing the cursor, and the "has()" function returns false if the cursor is at the end of the line. This structure deals with tokenizing, ignoring white spaces and skipping comments so we don't have to deal with them again in our code.

Second tool we needed was a function, "expressionParser", that dealt with expressions which we might encounter in many parts of the input code. We decided to write a function that implements the Shunting-Yard algorithm for infix-to-postfix conversion. The algorithm was missing some functionalities such as syntax checking of the infix expression and an easy way of implementing the "choose" function, but was easy enough to modify. We also decided to write an accompanying function that implemented the "choose" functionality. This function uses an algorithm that counts the open and close parentheses to determine which comma belongs to the choose that is being read, since it could also belong to a nested choose function. Using this, the choose expression is divided into four expressions whose results are then written into "select" and "icmp" expressions in llvm code to attain the desired functionality. At the end we ended up with a function that can take an expression as its input, write all the calculating code to the output file, and simply return the id of the temporary variable that will hold the result of the calculation.

The third general tool is another structure that handles the declaration of variables, "variableHandler". Since the variables can be declared anywhere in the code, but need to be properly allocated at the beginning of the IR code, we felt the need to have a structure that will store them and check for their validity.

The most important part of the code is the main function that does all the translation. The main function assigns the file that was passed to the program as an argument to an ifstream, then creates an intermediate file to write all the code to. An intermediate file is used because the later parts of a program can influence the earlier parts due to variable declarations. We decided to use an intermediate file for this purpose instead of an string because a string could pose some issues if the input code is long enough. Then a loop starts reading the file line by line, as long as the program hasn't encounter an error. We then get the first word and pass it to an if-else chain. We first check if the first word is a keyword, such as "while" and "if". If it isn't a keyword, and it is a valid id for a variable, the statement is treated as an assignment statement. If it is the "print" keyword, the expression inside the keyword is passed to the expression parser and the returned temporary variable is inserted to a copy of the llvm print code that we were given in the example. If it is a keyword that requires branching statements, the relevant expression is extracted and passed onto the expression parser, and then the required "br" statements are written to the output. The label for that particular conditional is then pushed to a stack in order to be used when we encounter the end of the block. The check for nested "while" and "if" statements are also done here, by checking if the label stack is empty. For the end of these statements, a check for the '}' character is added to the main if-else chain, in which case the label is popped from the stack and the required "br" statement is written to the output file. Most of the error handling is also done in the main function. For error handling we chose to use a boolean variable that is assigned a true value upon encountering an

Alperen Bırçak 2018400006
Çisel Zümbül 2018400093

error, which prompts the program to write out an error instead of some erroneous code. The syntax is checked as it is being read other than the errors that exist in expressions which are checked within the expression parser function. We chose this approach to not go over the code more than it is needed. Then at the end, if there were no errors, the boilerplate code and allocation statements are written to the real output file, and the contents of the intermediate code is copied over.

Our code accomplishes the given task without problem, but it could be improved. Even tough we tried to use a general approach to the problem some duplicate code still exists. Also the implementation of the choose function in the Shunting-Yard algorithm could be more elegant, we could integrate it into the algorithm instead of patching it over. The error handling also could be improved by using try-catch statements instead, since we decided to go for a more familiar option due to time constraints.