Scanner:

1. Dont care about uppercase and lowercase for token. => done. How to do: When reading the identifier in the readIdentifierKeyword(), use toupper for each character.

2. Get error when the number is too large => done. How to do: create a new error, and in readNumber, count the total number. If it reaches over the limit => give error

3. Create a new comment type aka // in C => done. How to do: in the case CHAR_SLASH, create a new token for slash operator. Then read a new char, if it's slash again => comment. If not => slash operator.

4. Identifier if longer than k characters then only take the first k characters => done. How to do: when reading the identifier, after storing each char, count it up til k, after k we dont store in the string anymore.

5. Allow one or some specific chars other than digit and letter in the identifier => done. How to do: look up the ascii table, add the corresponding char in the charcode.c, and adjust the readIdent function so that it also reads those characters.

6. Read char constant as doublequotes => done. How to do: Simply create a new charcode according to the ascii table. Then create a new function, loop with readChar() through those doublequotes and count the total number. If it matches the requirement number, then check similarly to singlequote. After checking, do the loop again. Remember to check for the EOF case.

Parser:

1. Declare multiple variables with the same data type. Example: var a, b, c, d, e : integer; => done. How to do: fix the compileConst/Type/VarDecl so that it calls itself recursively after eating the identifier, and the seperator that is defined (example: SB_COMMA). Remember to check for errors, if it's SB_EQ then stop calling recursively. If it is neither SB_COMMA nor SB_EQ => error

2. Dont care about orders when declaring const, type, var or function, procedure => done. How to do: In compile block, use a while loop that checks if tokenType is either const, type, var, function or proc and then call the corresponding compile funciton.

3. Remove the CALL symbol when calling procedures => done. How to do: if meet KW_CALL => invalid statement. In case TK_IDENT, since there are two cases that have TK_IDENT now (assignSt and procCallSt), so we need to eat(TK_IDENT) first, and then consider the next token type. If it's LSEL then compileAssignSt() without eating another TK_IDENT, but if there's no LSEL, then we need to check the follow set, which is the SB_ASSIGN symbol, if token type is SB_ASSIGN then eat(SB_ASSIGN) and compileExpression(). For the procCallSt case, if we meet LPAR, then compileArguments, and if we meet SEMICOLON (follow set) => we just simply break.

4. Two types of callSt for procedure, one has KW_CALL one does not. => done. How to do: The difficult part is to seperate two cases. Applying question 3, when we meet KW_CALL, we call the modified version of compileCallSt() like in the question. When we dont meet KW_CALL, then check for the condition of compileAssignSt() first. The default case will be the modified compileArugments(). Remember since we need to eat the TK_IDENT right away in order to identify the next token type, so in the compileCallSt() or compileAssignSt(), we dont

eat(TK_IDENT) anymore.

5. Arguments of functions / procedures must have () => done. How to do: just follow the rules to implement as this one is just a subproblem of question 4. No follow set needed.

6. add repeatSt and fix ifSt => done. How to do: just simply follow the rules as Condition does not have follow set. The follow set should be handled in the Expression that leads to Term which has been done already.