Language and Logic: Assignment #2

Helena Tevar

Problem 1

Design a Turing Machine that can perform unary arithmetic without parentheses. Your machine should support the following operations:

- Multiplication (x)
- Division (/)
- Exponentiation ()
- Modulo (%)

Solution
The Turing Machine for this exercise, shown in Figure XXX, follows the next description:

$$\Sigma = \{1, x, /, \%, \wedge\}$$

Σ does not contain the blank symbol, here called b. It is included in the tape alphabet.

$$T = \{1, 2, x, /, \%, \wedge, b\}$$

The start state is q_0 and the accepting state is q_{acc}.

The Turing Machine works by concatenating different and smaller Turing Machines. The first one show in Figure 1 find the first operator and send the input to the sub machine that represents that operator. For instance, if the machine finds a module operator, it will send the input to the module machine. All the machines that work with an operator will write the solution at the right of the tape after a blank space. Once the operator machine has ended, a rearrange machine in q_{500} will fix the tape order, moving the left overs to the right of our result. For instance:
Input Tape: 11X11/11
Multiplication Solution: /11 1111
Rearrange Machine Solution: 1111/11
The "rearrange machine" (Figure 6) will send its solution to the first delivery machine and loop through all the calculations. Once the first delivery machine finds only unary numbers and none operators, it will accept that unary number as the final solution and transition to the accepting state.
The module operation ends up already fixed, so it is no need of send the solution to the "rearrange machine" and will go through the "delivery machine".

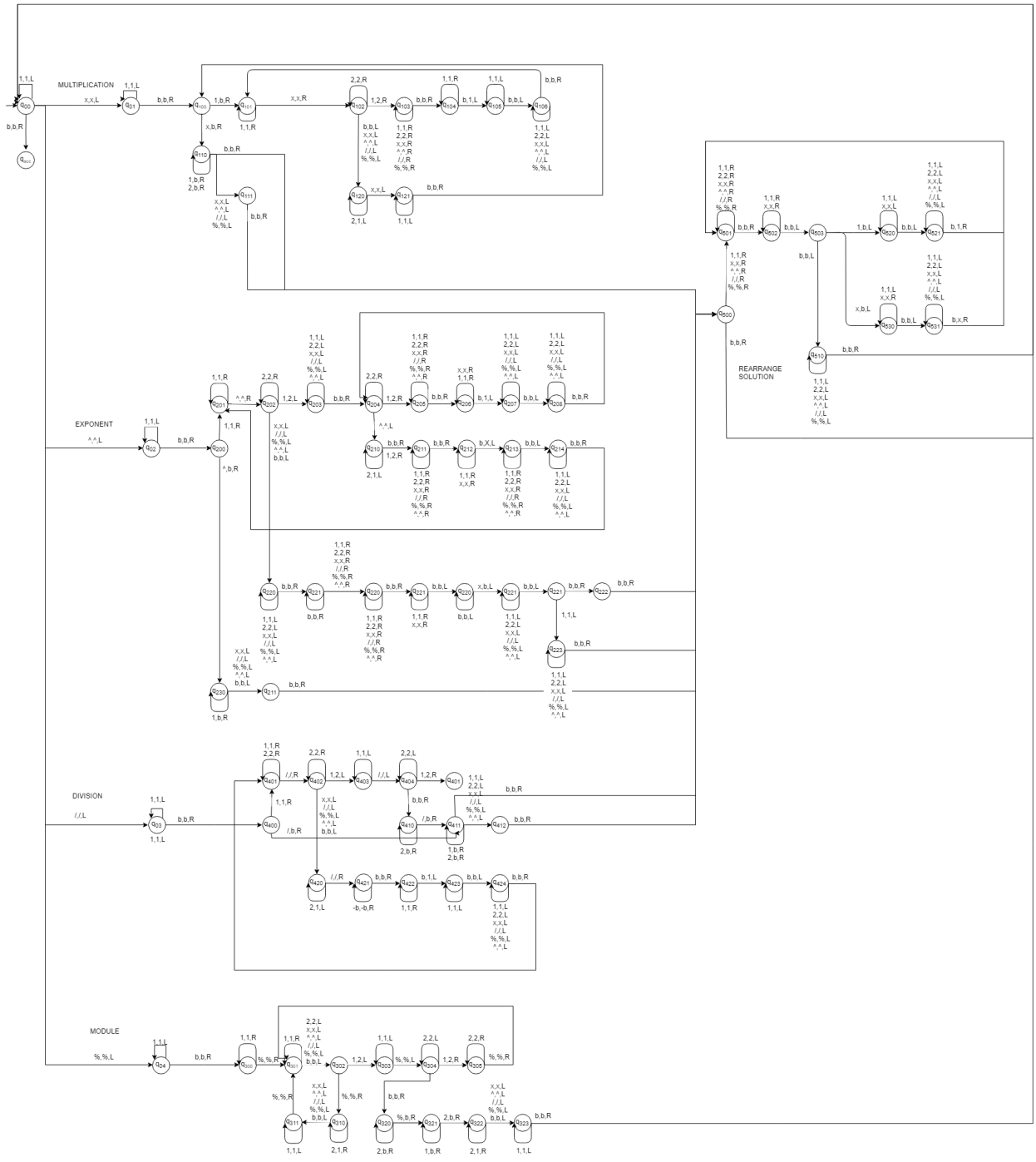

Figure 1: Turing Machine

Multiplication Machine

The multiplication machine are all those states inside the group of q_{100} to q_{121} and ends in q_{500}, that is the "first" state of the "rearrange machine" that will fix the tape to be able to loop again.
q_{100} has to possible transitions, when reading 1 or x from the tape. When finding 1, the machine will work by deleting numbers from the multiplicand, changing the multiplier 1 for 2 to spare the count, adding a 1 to the result and returning to the beginning, changing the 2 to 1 in the process. If the machine finds an X

in q_{100}, then the multiplication is finished and will proceed to delete the x and the multiplier.
In Figure 2 it can be seen the purple color for the state that deletes the multiplicand, yellow for the part that changes the multiplier 1's for 2's, red for the process of writing the solution at the end of the tape, green for the part that changes the multiplier 2's by 1's and blue for the part that cleans the multiplier and send the result to the "rearrange machine" starting in q_{500}.

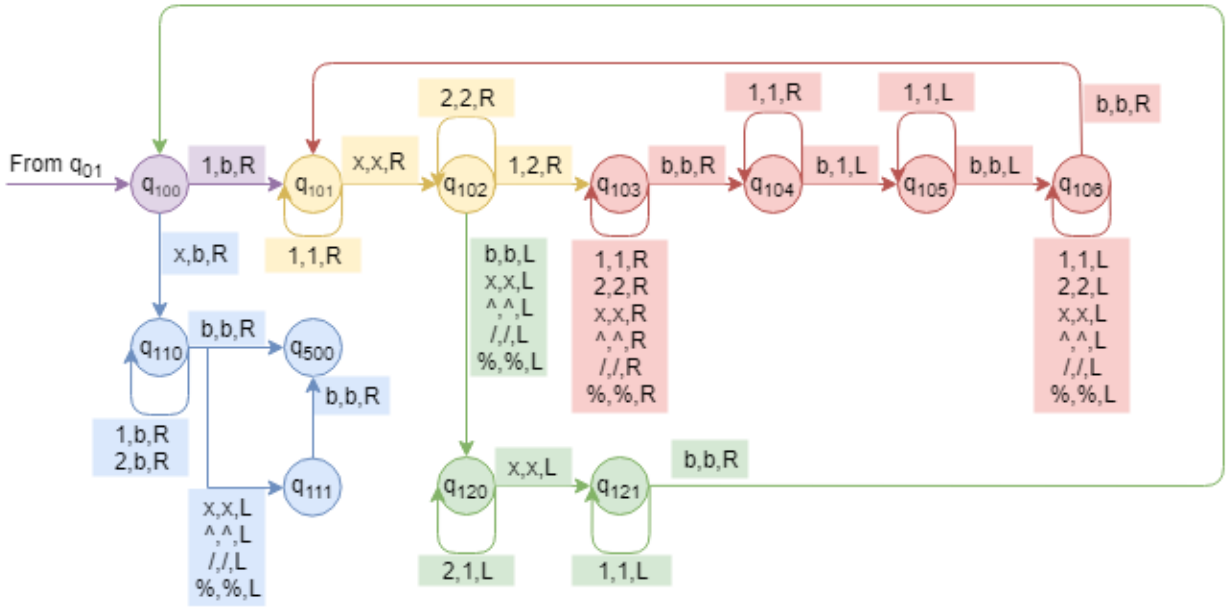

Figure 2: Product Turing Machine

Division Machine

This machine will check if the first element is equal to zero, that means when the module of the division or the actual dividend is zero. If is not, the machine loop around dividend and divisor, changing 1'2 to 2's as counters. If the divisor has no 1's left, the divisor will reset the 2's into 1'2 and add 1 to the solution. This process will loop until the machine finds zero in the dividend. When dividend is zero, the machine will delete the formula and go to the "rearrange machine".
Inf Figure 3 Can be seen that the yellow part is the counter part. The red states reset the dividend and start again the division. The blue states clean the first and second element and finish the operation.

Problem 1 continued on next page...

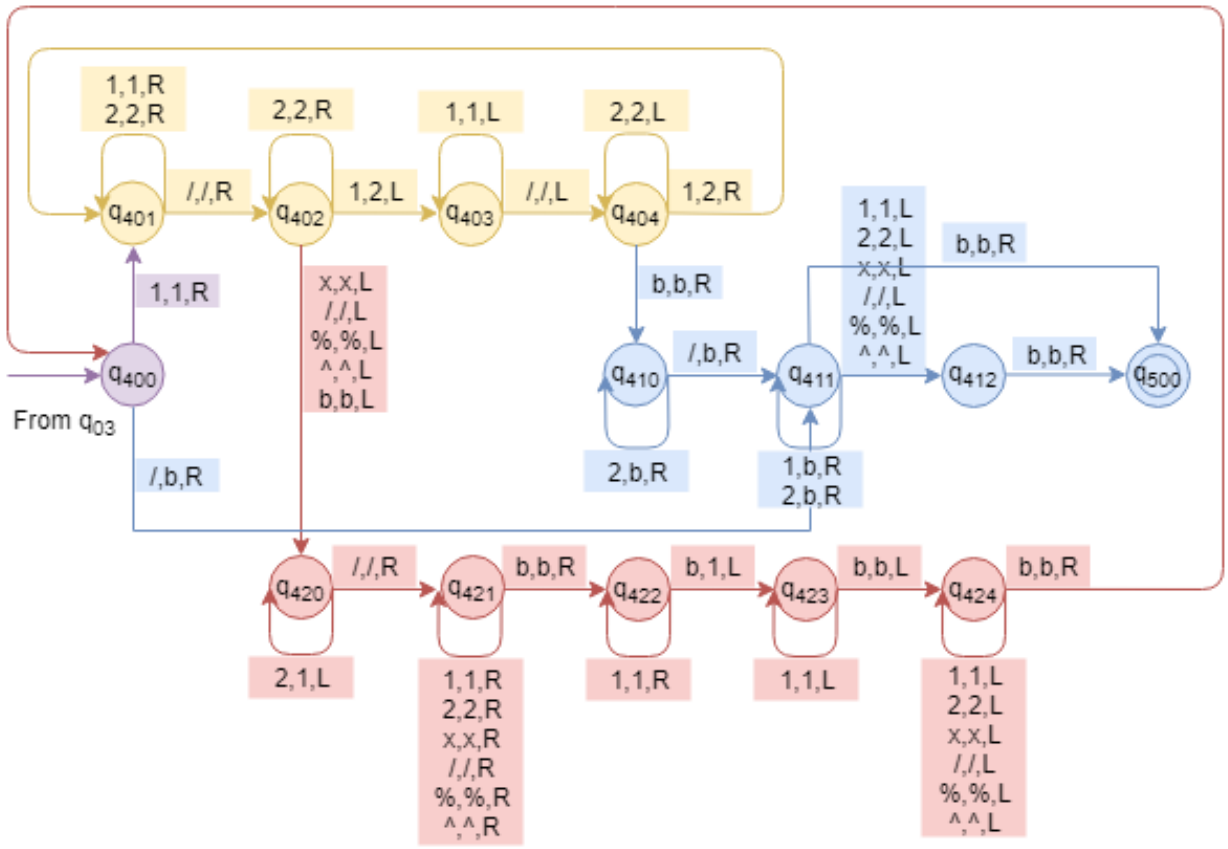

Figure 3: Division Turing Machine

Module Machine

Similar to the division, it makes a division with a dividend and divisor. The difference is that a division counts how many divisors have we taken from the dividend, and here we are going to count how many dividend we have left. We you try to subtract the divisor from the dividend you find out that there are not enough units in the dividend to remove a whole divisor. At this point you should have marked some of the units of the divisor. The amount of marked units in the divisor equals the module, with the remarkable exception of one of them, because we marked one of the module units before checking if there is some dividend units to remove.

The process, as said, stops when there are no more units in the divisor to remove and the solution of module is ready just need to clean the leftovers. Therefore we remove all the marked units in the dividend, all the 2's, and the module operator and all 1's form the divisor and one of the 2's that we marked before checking if there was any element in the dividend. After that, when there is left some 2's or none (module zero) the machine changes 2's to 1's, leaving the result before the next operation (or not). That is the reason why there is no need of a rearrangement after we finish the operation.

The Figure 4 shows in purple the states that look for the divisor, yellow are the states that bounce between divisor and dividend, red are the elements that reset the divisor and blue states remove the leftovers and leave the resultant 2's to 1's.

Problem 1 continued on next page...

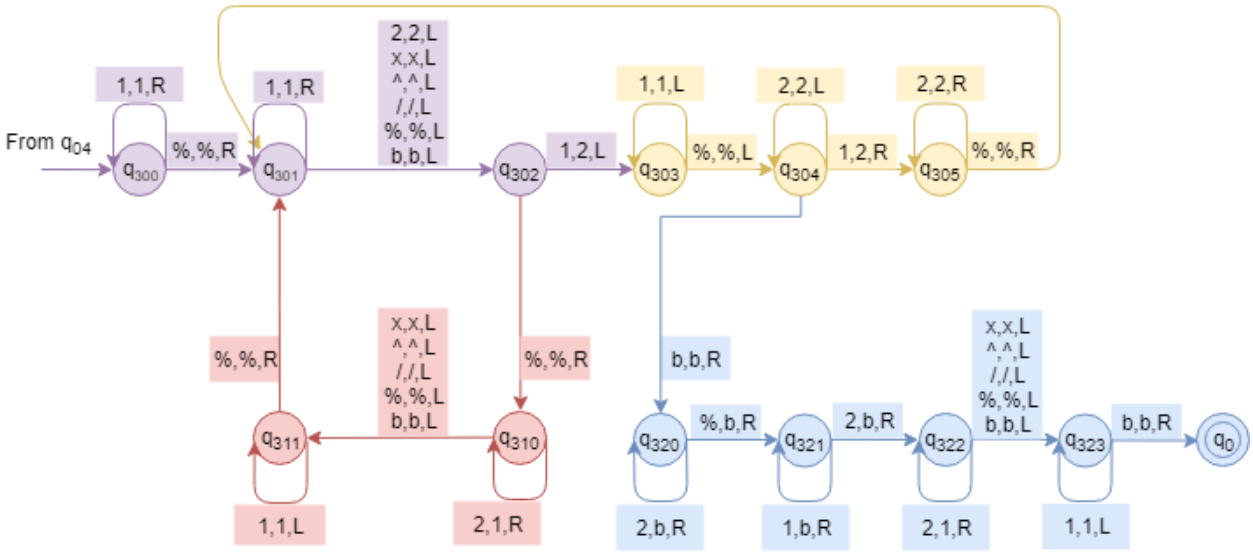

Figure 4: Module Turing Machine

Exponential Machine

This machine does not provide exactly a solution but a multiplication. Exponential has two "Go To q500" state because it counts when the base is zero. The machine will work (when base is not zero) by using the exponent as a counter to repeat the base and add a symbol x at the end of the process. The machine works but creates an extra x symbol at the end, so it is fixed before reaching the state q_{500}
In Figure 5 is shown how the purple states mark the exponent and goes to the base and start marking 1's to 2's. Yellow states will copy the base to the end of the tape. Red states reset the exponent and add an x to the end. When there is no more exponents, the blue states fix the extra added x at the end of the solution and goes to q_{500}

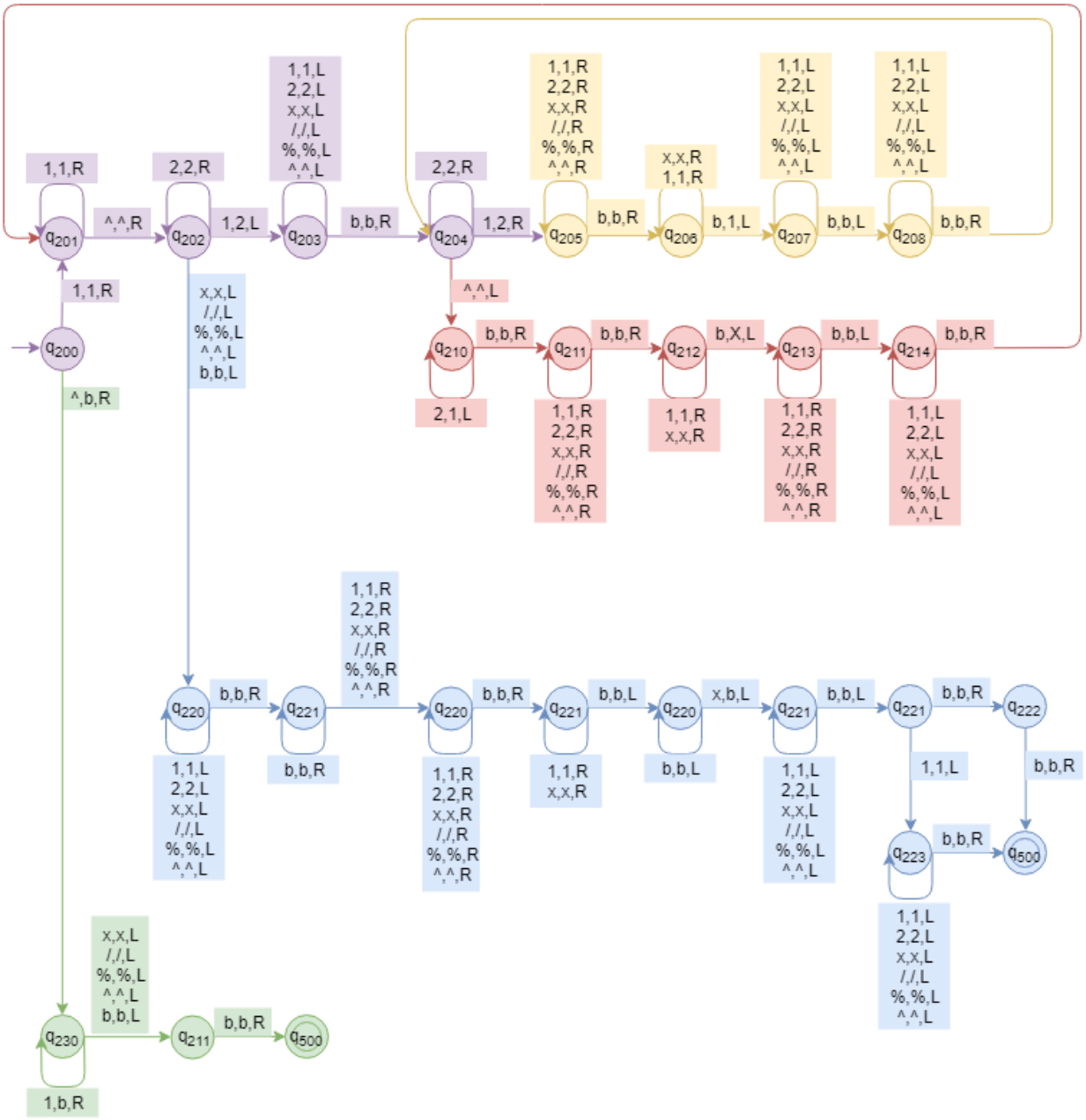

Figure 5: Exponent Turing Machine

Rearrange Machine

This machine will take all the elements from the solution, delete them and copy them at the left of the next operation.

The machine, Figure 6 purple states checks if there is anything in the tape or if is empty (for example, a division resulting in zero). If there is something in the tape, the machine will look for the first blank and the possible element behind it (Yellow states). If there is none, the the machine will go to the first state and loop again(Blue states). If there is some element, like a 1 or a x (see exponential machine), then the elements will be deleted and written at beginning of the tape (red states).

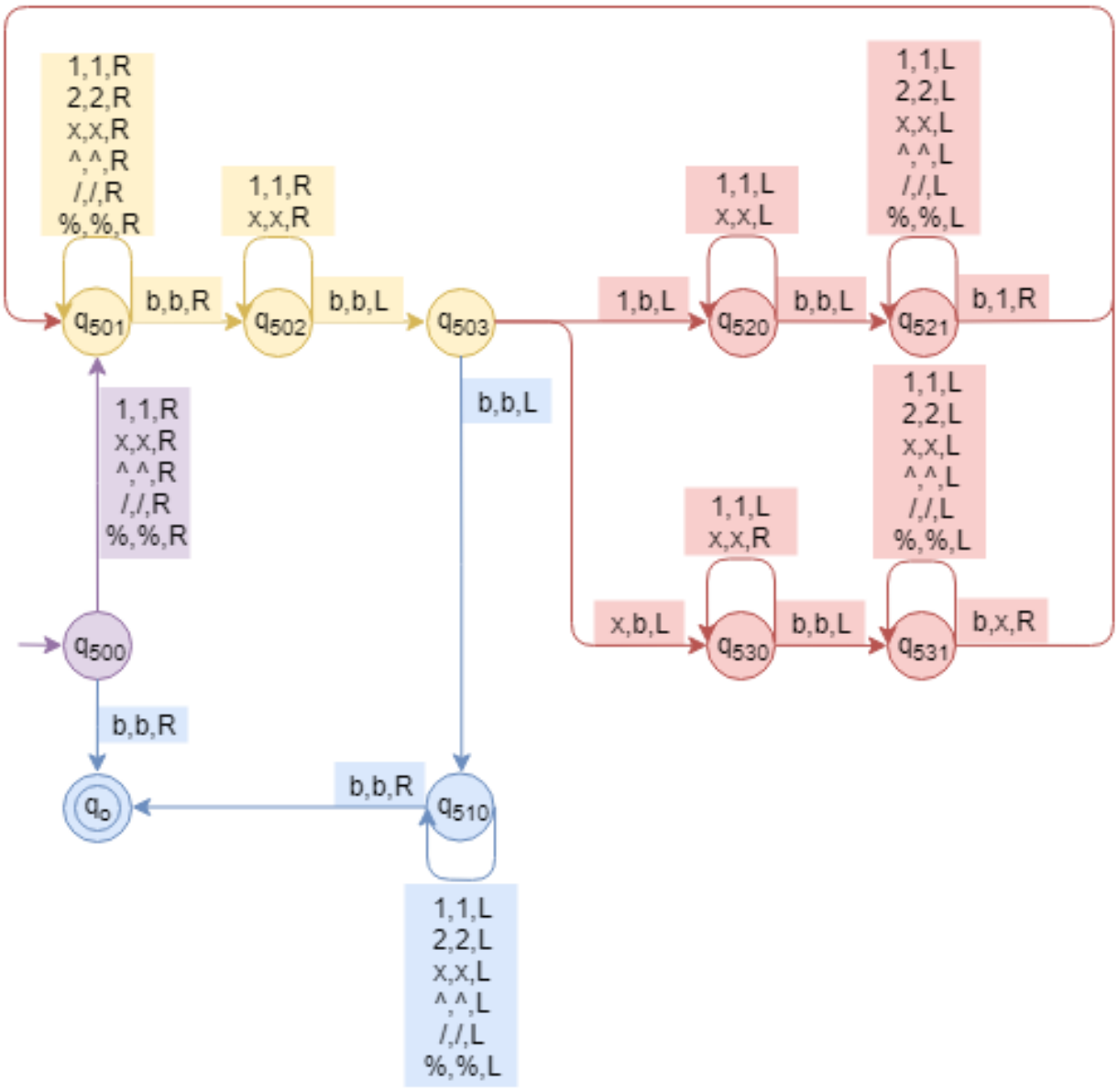

Figure 6: Rearrange Turing Machine

Problem 2

Context Free Grammar

The grammar has been created using a recursive pattern that is followed in Atnlr4 for Eclipse - Windows. During the creation of the rules I encountered the same loops, that while working with programming languages as Java, would be easy to solve, here was more complicated. Anyways, the loops are used pretty often and the only difference is the rule that they are working with.In order to make the grammar below clearer to understand, this are the common patterns followed for the loops of different rules.
Rule: A $\rightarrow$ Rule
Zero or more

$$A^* \rightarrow AA^*|\epsilon$$

Zero or one

$$A? \rightarrow A|\epsilon$$

One or more

$$A+ \rightarrow aA + |\epsilon$$

I included different trees in this exercise because it was impossible to create one single tree and make it visible in LaTeX. The tree is not to prove a string formally but to understand informally the complexity of the grammar. It has many leaves so it is no ambiguous, by creating different rules for different parts of the strings.

Grammar

The starting rule that will call further rules is called "r". The first step of this grammar would lead to the *Struct* rules or the *Func* rules, as seen in Figure 10. This grammar allow the creation of different structures that are mostly divided in the initial declaration of the structure, a block of curly brackets and the content of it, that would be a loop of declaration of variables.

Problem 2 continued on next page...

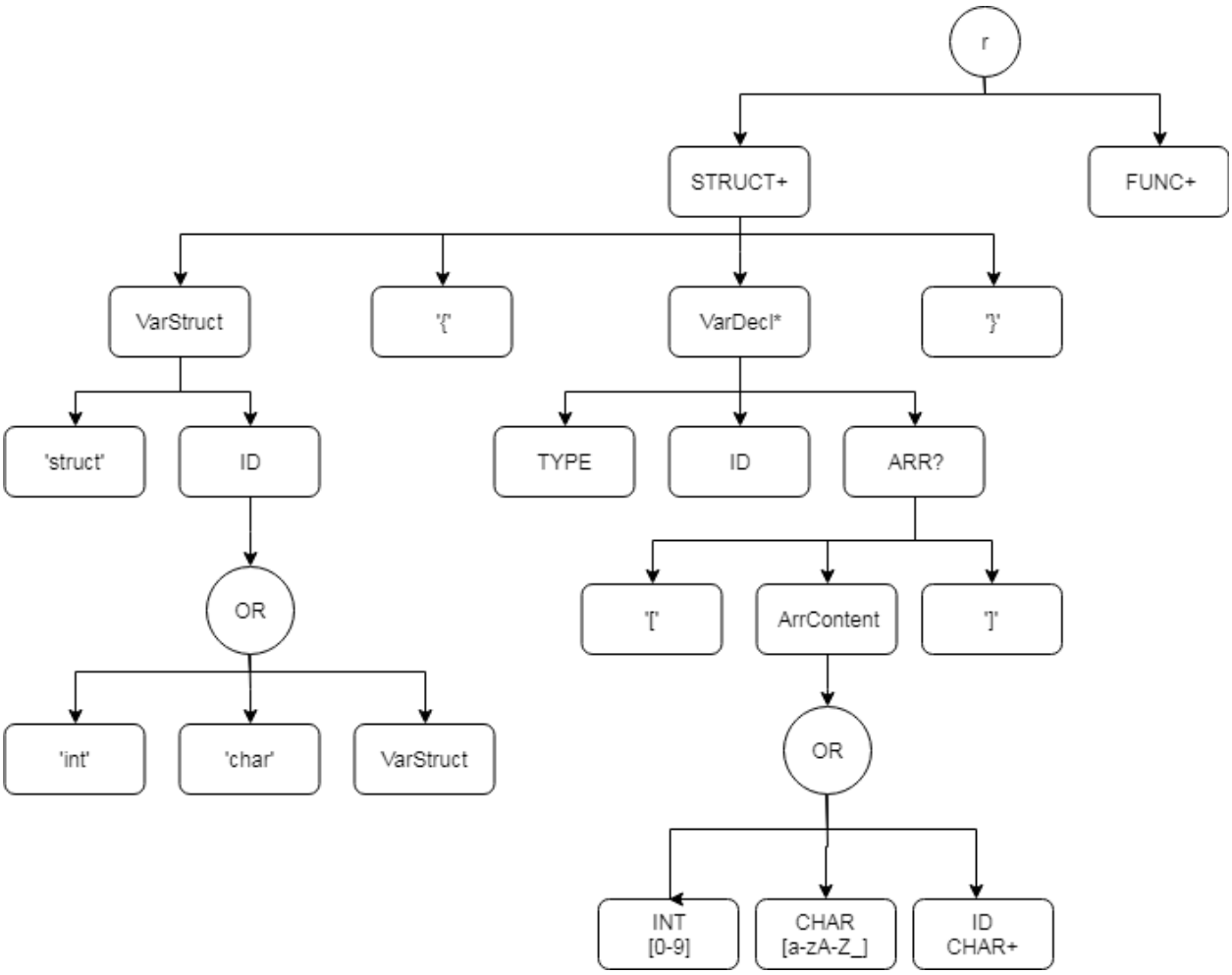

Figure 7: Grammar tree - Struct

The next part of the grammar gets more complicated. The grammar of *Func* includes four branches. The first leaf will rule over the first two words of *Func*, the second leaf controls the parameters, the third rule controls the block between curly brackets in this specific place and the last leaf will control that there is a return and end of the block.The parameter rule will control everything that is inside parentheses, including a loop of multiple possible arguments or none at all.

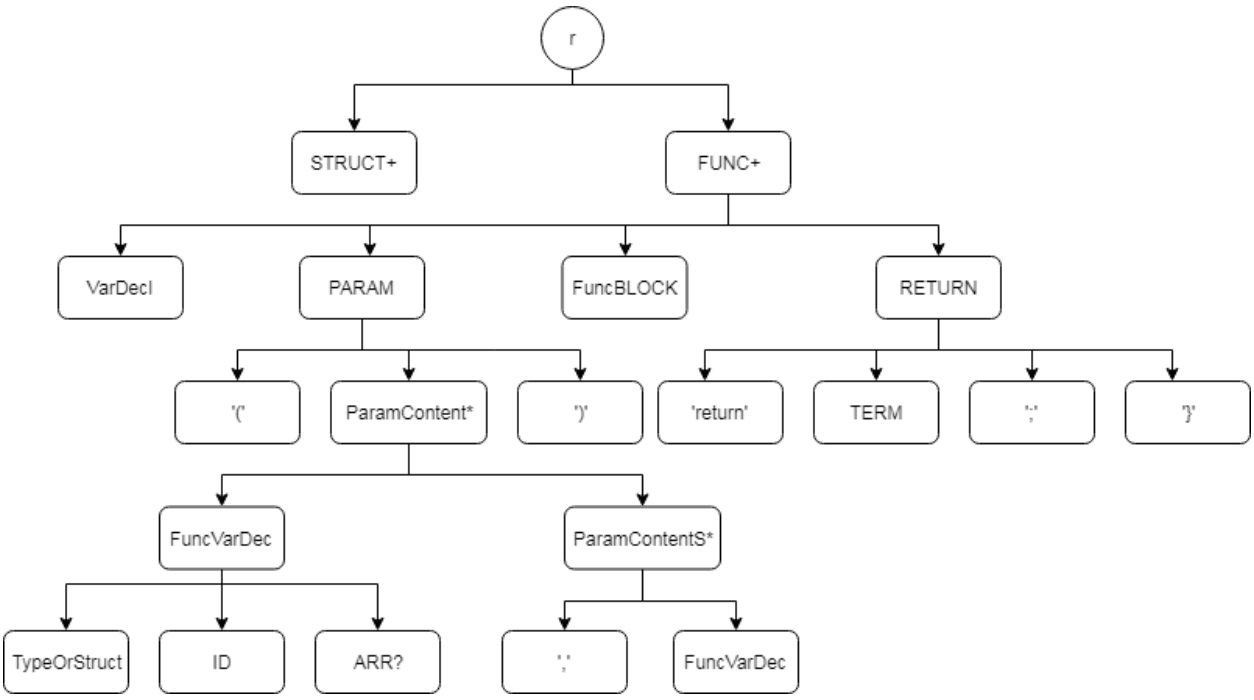

Figure 8: Grammar tree - Func Part 1

The block after the parameter will control two possibilities. That the content includes a string with arguments between parentheses, for instance move_snake(snake,direction) or a variable with a type. This option also includes arithmetics, using unary operators.

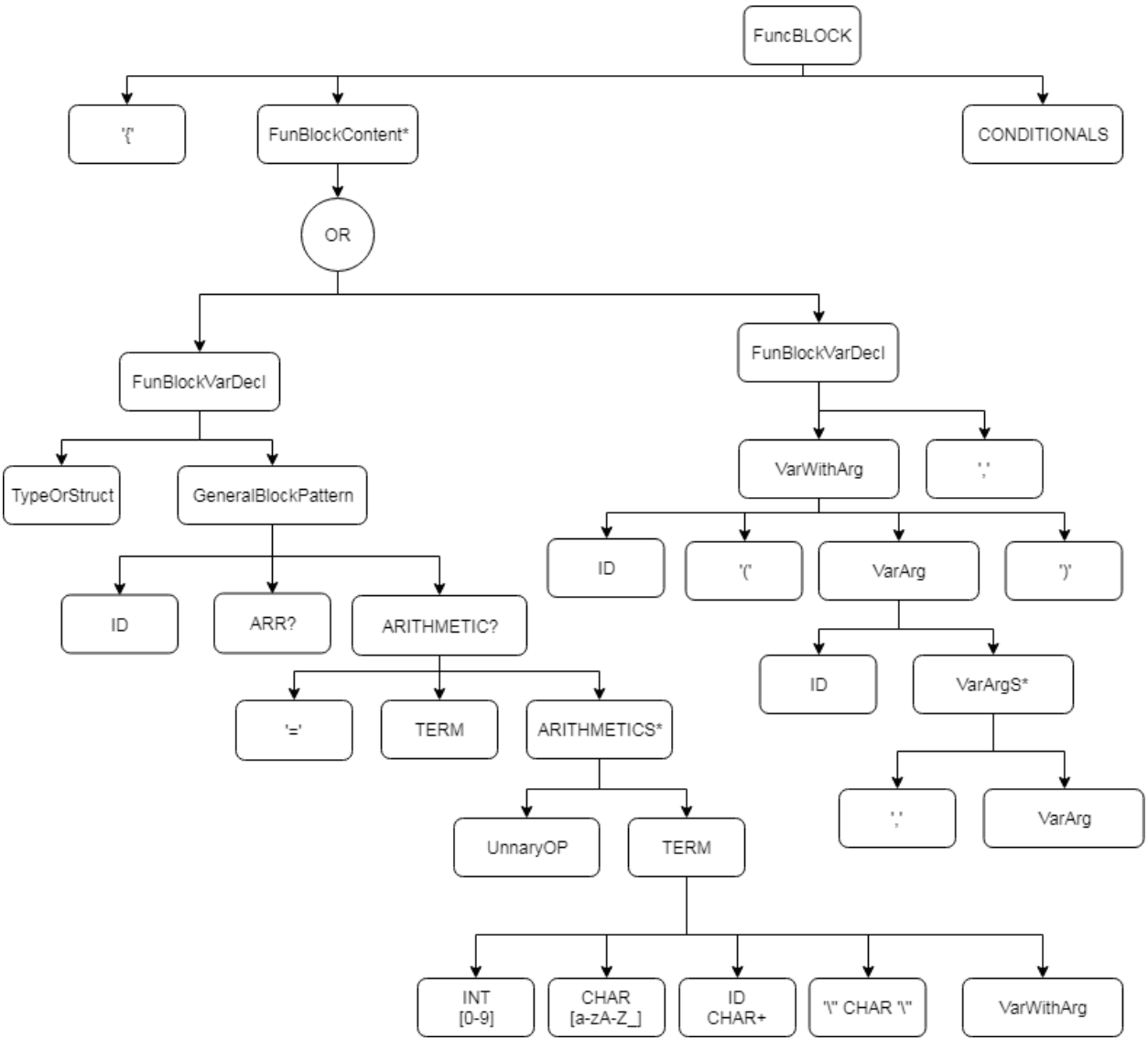

Figure 9: Grammar tree - Func Part 2

This block may also contain conditionals (and conditionals inside conditionals). Conditionals are created with a condition declaration, for instance "while" or "if", a parameter with a conditional expression between parentheses and a block. The variables inside this block were no exactly the same as in other kind of blocks, so in order to avoid any kind of ambiguity I created a special type of variable for this instance. The content of the block in conditionals may content different variables and more conditionals. Here it is used a special rule, called GeneralBlockPattern that reunite common strings found in both the conditional block and the Func block.

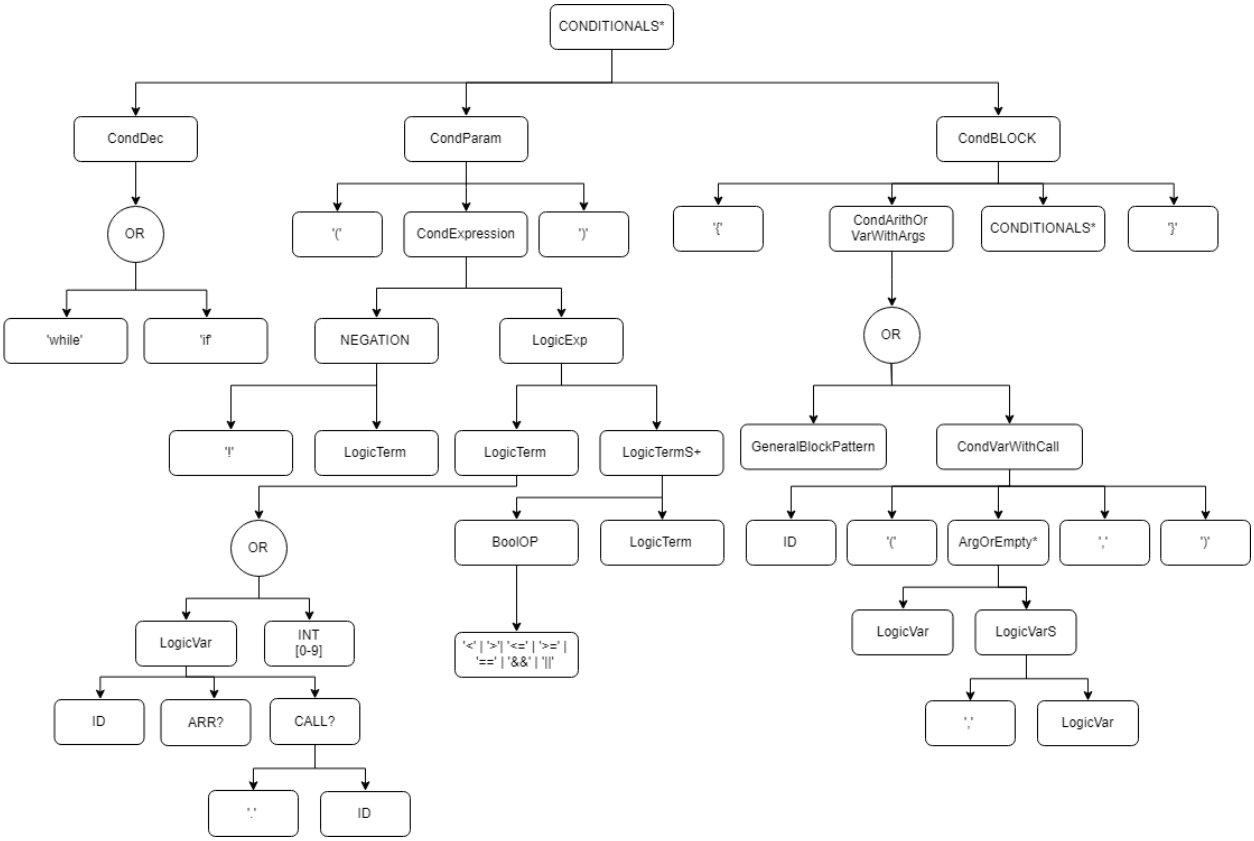

Figure 10: Grammar tree - Func Part 3

That is all the explanation about the tree created for this occasion. If we were to test the tree with a string, we will see that the different blocks created for each need encapsulate pretty well the possible similarities.

```
int main (char foo) {int bar; if (foo && bar) {foo (bar);} return foo;}
```

If we test the string above in the tree, we find it impossible to create using two different paths. At the moment we reach a parentheses, the inside has it own branch. Same happens for the parentheses in the conditional and the block created. They have their own branches inside the tree, so it is not confused with any other option.

Actual Rules

1. r → STRUCT+ FUNC+
2. INT+ → [0-9]
3. INT → INT+ INT $|\epsilon$
4. CHAR → $[a - zA - Z_]$
5. ID+ → CHAR |
6. ARR→ "[" ArrayContent "]"
7. TYPE→ "int" |"char" |VarStruct

8. VarStruct→ "struct" ID
9. ArrayContent→ INT |CHAR |ID
10. ARR?→ ARR |ϵ
11. STRUCT→ VarStruct "" VarDecl "" ";"
12. STRUCT+→ STRUCT STRUCT+ |ϵ
13. VarDec→ TYPE ID ARR?
14. VarDec*→ Vardecl ";" Vardel* |ϵ
15. FUNC→ VarDec PARAM FunBLOCK RETURN
16. FUNC+→ FUNC FUNC+|ϵ
17. PARAM→ "(" ParamContent ")"
18. ParamContent*→ ParamContent |ϵ
19. ParamContent→ FuncVarDec ParamContentS*
20. ParamContentS*→ ParaContents ParaContentS* |ϵ
21. ParamContentS→ "," FuncVarDec
22. FuncVarDec→ TypeOrStruct ID ARR?
23. TypeOrStruct→ TYPE |STRUCT
24. FunBLOCK→ "" FunBlockContent* CONDITIONALS* ""
25. FunBlockContent*→ FunBlockContent FunBlockContent* |ϵ
26. FunBlockContent→ FunBlockVarArg |FunBlockVarDecl
27. FunBlockVarArg→ VarWithArg ";"
28. VarWithArg→ ID "(" VarArg? ")"
29. VarArg?→ VargArg |ϵ
30. VarArg→ ID VarArgS*
31. VarArgS*→ "," VarArg |ϵ
32. FunBlockVarDecl→ TypeOrStruct GeneralBlockPattern
33. GeneralBlockPattern→ ID ARR? ARITHMETIC? ";"
34. ARITHMETIC?→ "=" TERM ARITHMETICS*
35. ARITHMETICS*→ ARITHMETICS ARITHMETICS* |ϵ
36. ARITHMETIC→ UnnaryOP TERM
37. UnnaryOP→ "+" |"-" |"*" |"/"
38. TERM→ INT |CHAR |ID |" CHAR " |VarWithArg

39. CONDITIONALS*→ ConDecl CondParam CondBLOCK
40. ConDecl→ "while" |"if"
41. CondParam→ "(" CondExpress ")"
42. CondiExpress→ Negation |LogicExpression
43. Negation→ "!" LogicTerm
44. LogicTerm→ LogicVar |INT
45. LogicVar→ ID ARR? CALL?
46. CALL?→ CALL |ϵ
47. CALL→ "." ID
48. LogicExpression→ LogicTerm LogicOperation+
49. LogicOperation → BoolOP LogicTerm
50. LogicOperation+ → Logicoperation LogicOperation+ |ϵ
51. CondBLOCK → "" ConContent* CONDITIONALS* ""
52. ConContent→ GeneralBlockPattern |ConVar
53. ConContent*→ ConContent ConContent* |ϵ
54. ConVar→ ID "(" ArgOrEmpty ")" ";"
55. ArgOrEmpty→ LogicVar+ |ϵ
56. LogicVar+→ LogicVar LogicVarS
57. LogicVarS→ "," LogicVar LogicVarS |ϵ
58. RETURN→ "return" TERM ";" "}"