Linnéuniversitetet
Kalmar Växjö

Assignment 1:
Language and Logic

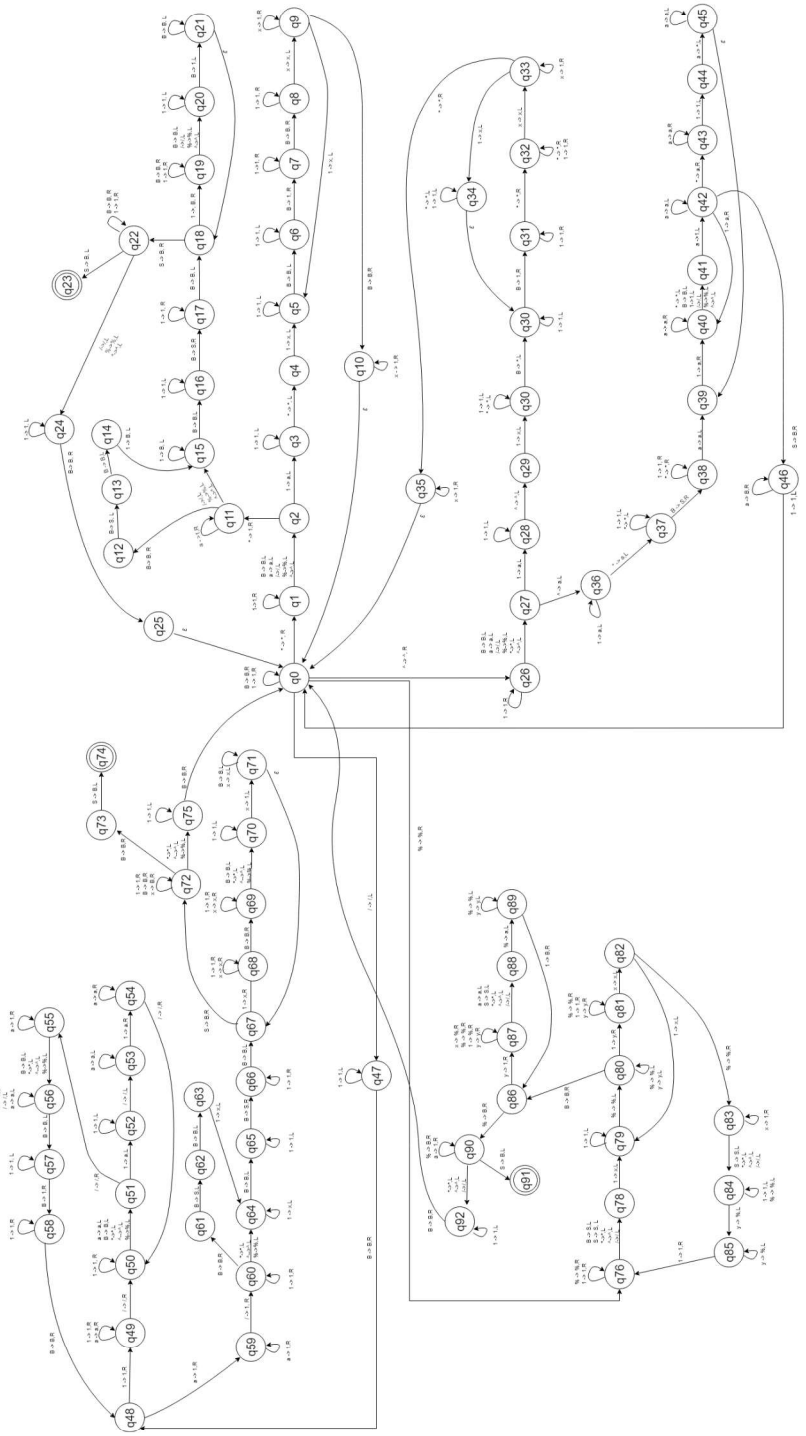
Ruth Dirnfeld
rd222dv@student.lnu.se

Faculty of Technology
Department of Computer Science

Table of contents

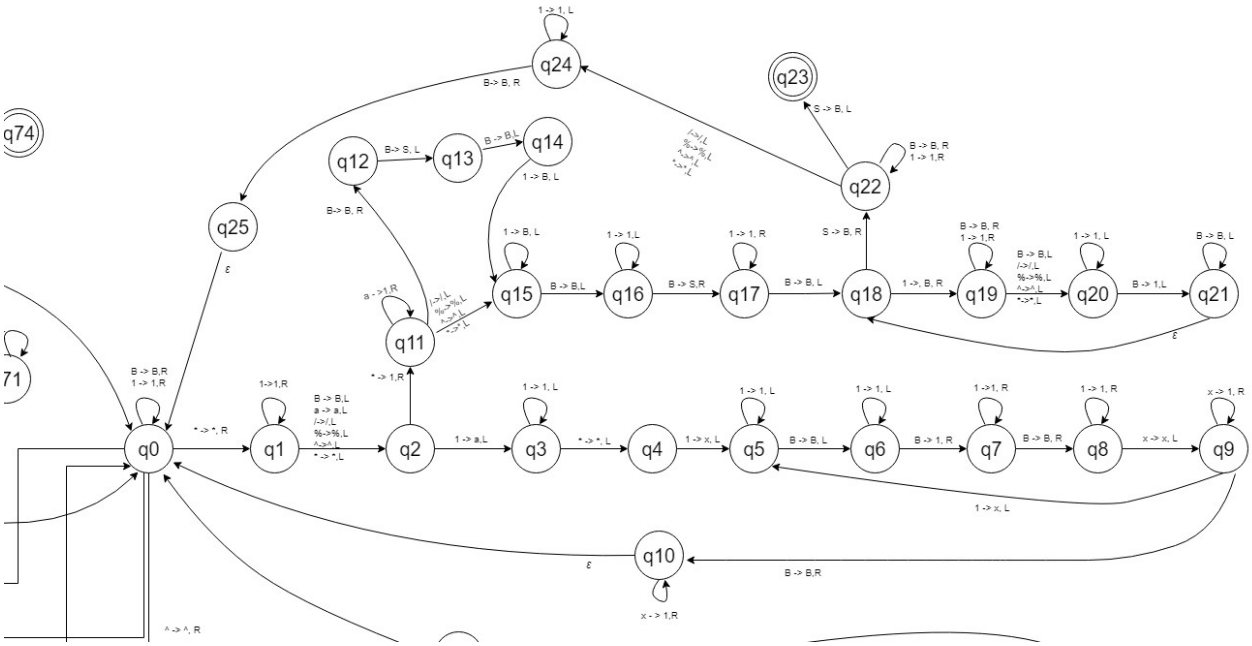
Faculty of Technology
Department of Computer Science

1. A Turing Machine that can perform unary arithmetic without parentheses:



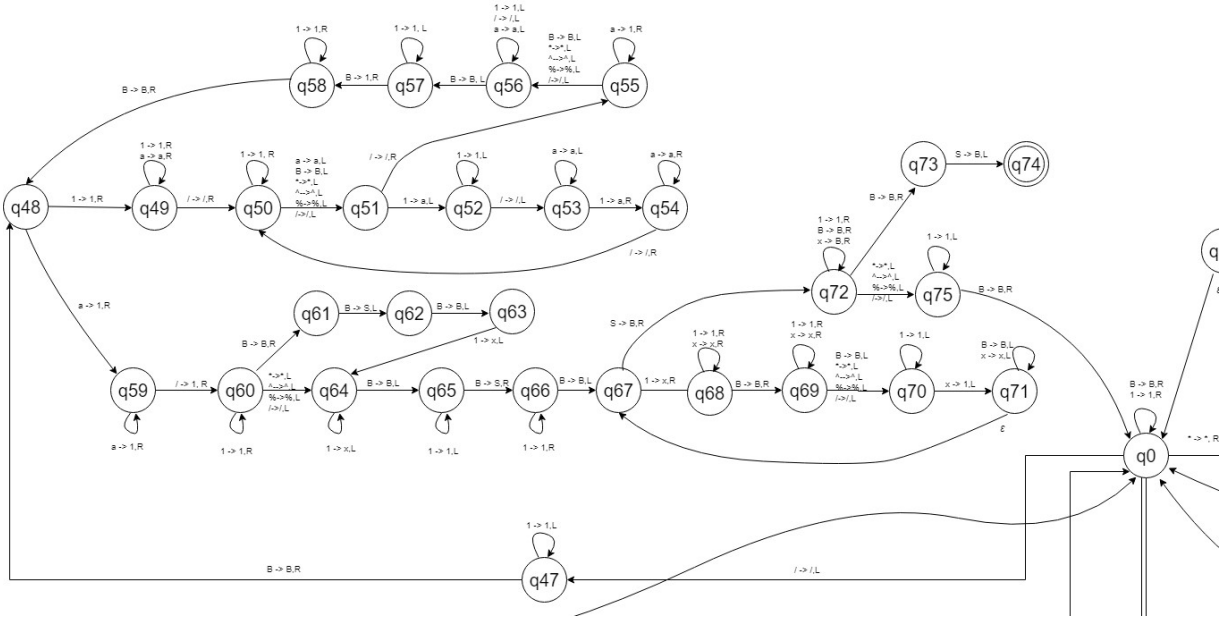
Faculty of Technology
Department of Computer Science

1.1 Multiplication (*) start>q0->q1->…



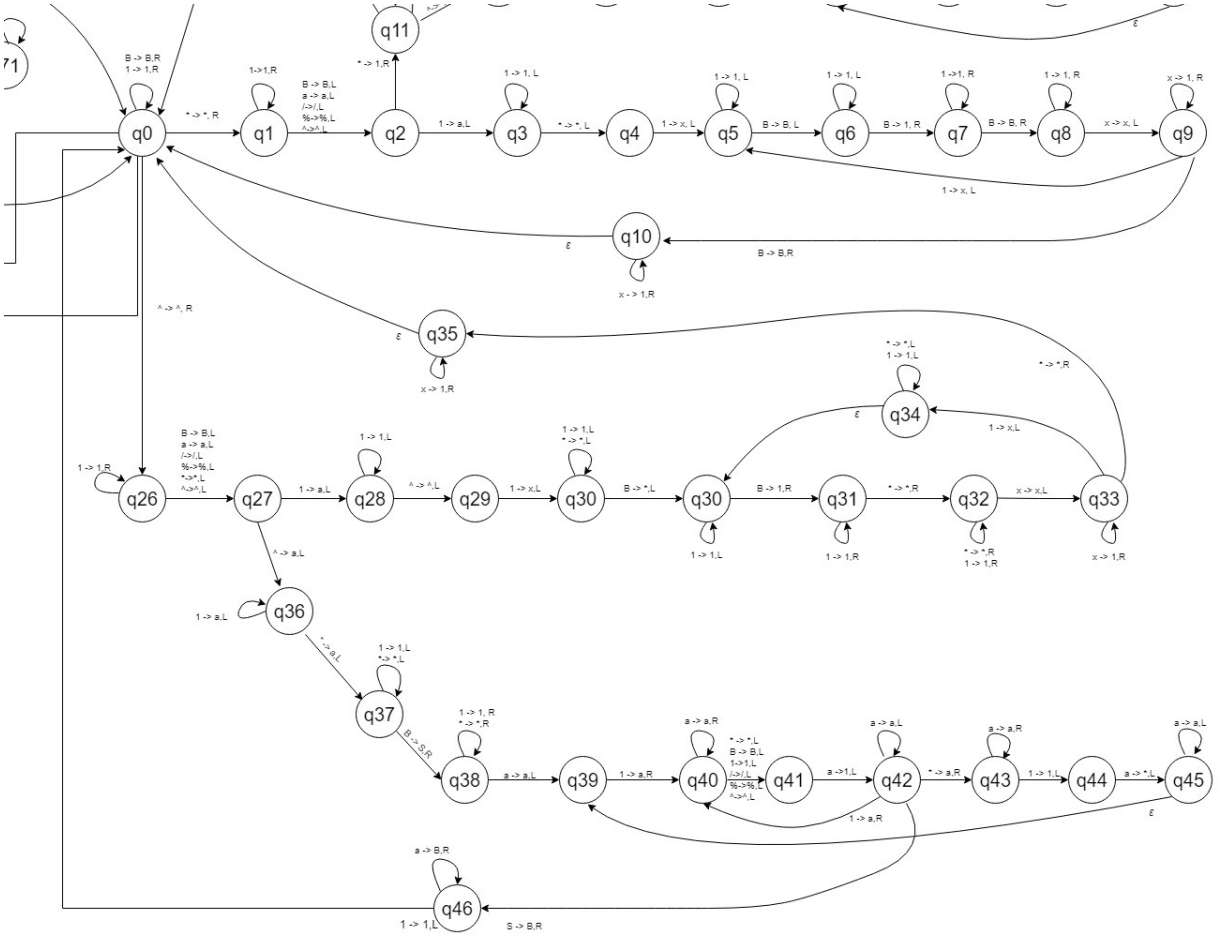
The above diagram is a zoom in of the multiplication diagram which performs multiplication like a Turing machine. For the explanation I will use B111*11E. I use the E only for the explanation, not in the above diagram. Here E={B,%,/,^,*}. Starting at q0, we loop right over all 1's and identify that we are performing multiplication. After performing a couple of left and right operations we start the multiplication and save the result to the very left side of the tape. So that the first part of the multiplication results in B111Bxxx*1aE. We are now in q9 and continue the same operations in order to turn the 1 on the right hand side also to an a and save the resulting multiplication to the very left side of the tape as well. After this is done, we are again in q9 with the current tape being B111111Bxxx*aaE. We turn the x's into 1's, continue with basic tape operations until we come to q2 and hit the '*'. Here we start turning the equation to B's and mark the very left side after the result with an S (q16-q17), in order to be able to identify where the end is. We then start the copying of the six 1's to the right side. After everything is copied, it depends if E is a B. If E is a B then, after we hit the right side S marker, we overwrite it with a B and we are done. If the E is another unary arithmetic operator, we get to q0 and start with the next part of the equation.
(PS: Watch out for the '*', it looks more like a dot at the top, but the '*' as it looks between q0-q1 it looks later on all diagrams the same)

1.2 Division (/)start>q0->q47->…



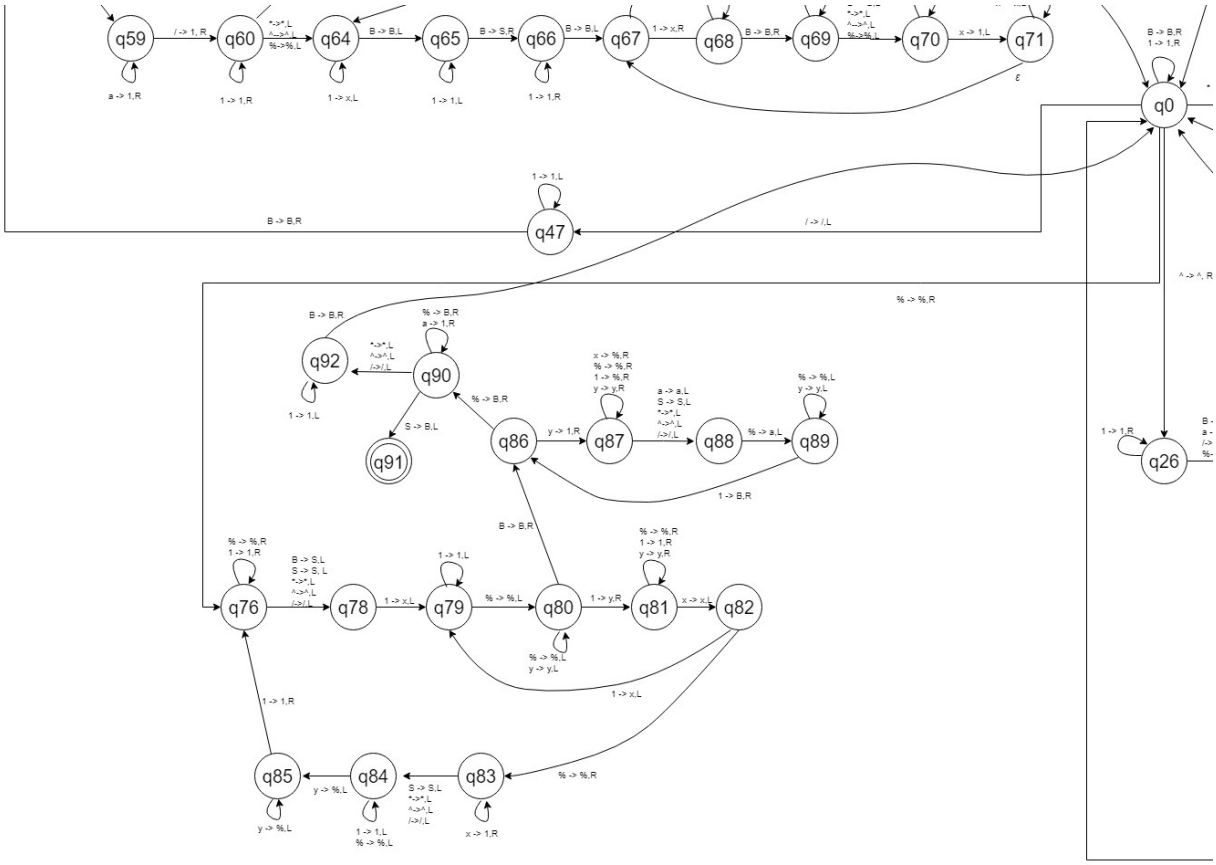
As an example for the division part, I take B111111/11E where E={B,*,%,^,/}. Similar as with the multiplication, we follow a couple of basic left and right operations and after the first division we end up in q51 with B1111aa/aaE. We have performed the first division and save the result to the very left side of the tape and end up with 1B1111aa/11E (on the way we turned the right hand side a's to 1's). From q48, we continue with the same approach and after the next division loop is done we end up first with B1B11aaaa/aaE and save again to the left and end up with B11B11aaaa/11E in q48. In this example, we do the same division one more time and end up in q48 with B111Baaaaaa/11E. In q48 we identify that we are done with the division as there are only a's left on the right hand side. We now start with the copying of the 1's to the right side and deleting the middle part. This is done in a similar way as in the multiplication, by marking the start and end with an S. At the end, we identify again if E is B or one of the operators *|%|^. If it is B, we are done and accept, if not, we move on to q0 and identify the next operator.

1.3 Exponentiation (^) start>q0->q26->…



As an example for '^' I take B1111^111E, where E={B,*,/,%,^}. Similar as in the other above diagrams, we loop to the most right side of the equation and set it as a counter. We take the four ones on the left hand side one by one and copy and save them on the left side of the equation. So that after the first cycle we end in the q0 with B1111*1111*1111^1aaE. We do this until we have no 1's on the most right hand side left. When we are done, we have converted 1111^111 to 1111*1111*1111 which we only need to send later to multiplication. So, as said before, when we have no a's on the most right hand side we can start blanking out and copying the converted version to the right side. This starts at q42->q43->… . In the Exponentiation we have no accepting state, since we send the conversion to the multiplication part.

1.4 Modulo (%) start>q0->q76->…



The above diagram shows the calculations if the next operator is % (mod). For example B111%11E where E={B,*,^,/}. After the first couple of operations the equation changes to B1yy%xxE. Continuing from q82 we proceed with the next operations and notice in q80 that we have no more 1's left and have currently By%%%xxE, so we start copying the result to the right sight. Similar as in the previous diagrams, we have a marked start and end of the equation and proceed by changing the very left y in the equation to 1 and the x's to %'s and end up with B1%%%%%E. We now change the first 1 to a % and the very right % to an a in q88 and end up with BB%%%%aE. Now we convert all %'s to B's the a to 1 in q90 and are done. If E is S, then we can accept and are done, if E is another arithmetic operator, then we go back to q and continue with the equation.

2 C sublanguage parser

2.1 Context Free Grammar for the language specification

A couple of explanations before the CFG:
EXP → VARDECL|NAME|STRUCTACCESS OP RHS|METCALL|NAME|CHAR SEMIorNOT

THE EXP rule takes care of all the Expressions according to the language specification. So for example, the CFG should cover:

(variable)=(functionname)(parameternames); // u=update(snake);
This is covered by NAME OP METCALL in the EXP rule

(astruct).(amember) // astruct.a=num;
This is covered by the STRUCTACCESS OP NAME in the EXP rule

Other examples that are covered by the EXP rule from the provided code are:
num=num*k; is covered by the NAME OP RHS in the EXP rule
c==’a’ for the ’a’, the CHAR rule had to be introduced
k>0 in order to be able to use the EXP rule also in the while (EXP), the SEMIorNOT rule was introduced

For handling STRUCT* (which means zero or many) separate rules had to be introduced. Taking STRUCT as the example, the rule MANYnoSTRUCTS → STRUCT|STRUCT MANYnoSTRUCTS| ε handels the zero or many option. This approach has been used in more rules as well.

NAME→<ASCII> are all ASCII characters, since I am not sure how to write them all out here without a Regex. If it was only letters I would write the rule: NAME → A | B | ... | Z | a | b ... | z

Context Free Grammar:

p → MANYnoSTRUCTS MANYnoFUNCTS

MANYnoSTRUCTS → STRUCT|STRUCT MANYnoSTRUCTS| ε

MANYnoFUNCTS → FUNCT|FUNCT MANYnoFUNCTS| ε

STRUCT → 'struct' NAME '{' VARDECLS '};'

VARDECL → TYPE NAME ARRorNOT

VARDECLS → VARDECL';'| VARDECL';' VARDECLS | ε

PDECLS → VARDECL','| VARDECL',' PDECLS | ε

PARAM → '(' PDECLS ')'

FUNCT → FUNCTDECL BLOCK 'return' TERM ';}'

FUNCTDECL → TYPE NAME PARAM

TYPE → 'int'|'char'|STRUCTS

STRUCTS → 'struct' NAME

STRUCTACCESS → STRUCTCALLER '.' STRUCTCALLER SEMIorNOT

STRUCTCALLER → '('|ε NAME ')'|ε

BLOCK → '{' EXPS WHILES IFS

EXP → VARDECL|NAME|STRUCTACCESS OP RHS|METCALL|NAME|CHAR SEMIorNOT

RHS → EXP|EXP RHS

METCALL → NAME PARAM

EXPS → EXP|EXP EXPS| ε

WHILE → 'while' '(' EXP ')' BLOCK '}'

IF → 'if' '(' EXP ')' BLOCK '}'

WHILES → WHILE|WHILE WHILES| ε

IFS → IF|IF IFS| ε

NAME → <ASCII>

INT → 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

CHAR → '\'' NAME '\''

TERM → INT | CHAR | NAME

ARRorNOT → '['INT']'| ε

AOP → '+'|'-'|'*'|'/'|'='

BOP → '||'|'&&'|'!'

COP → '<'|'>'|'=='|'<='|'>='

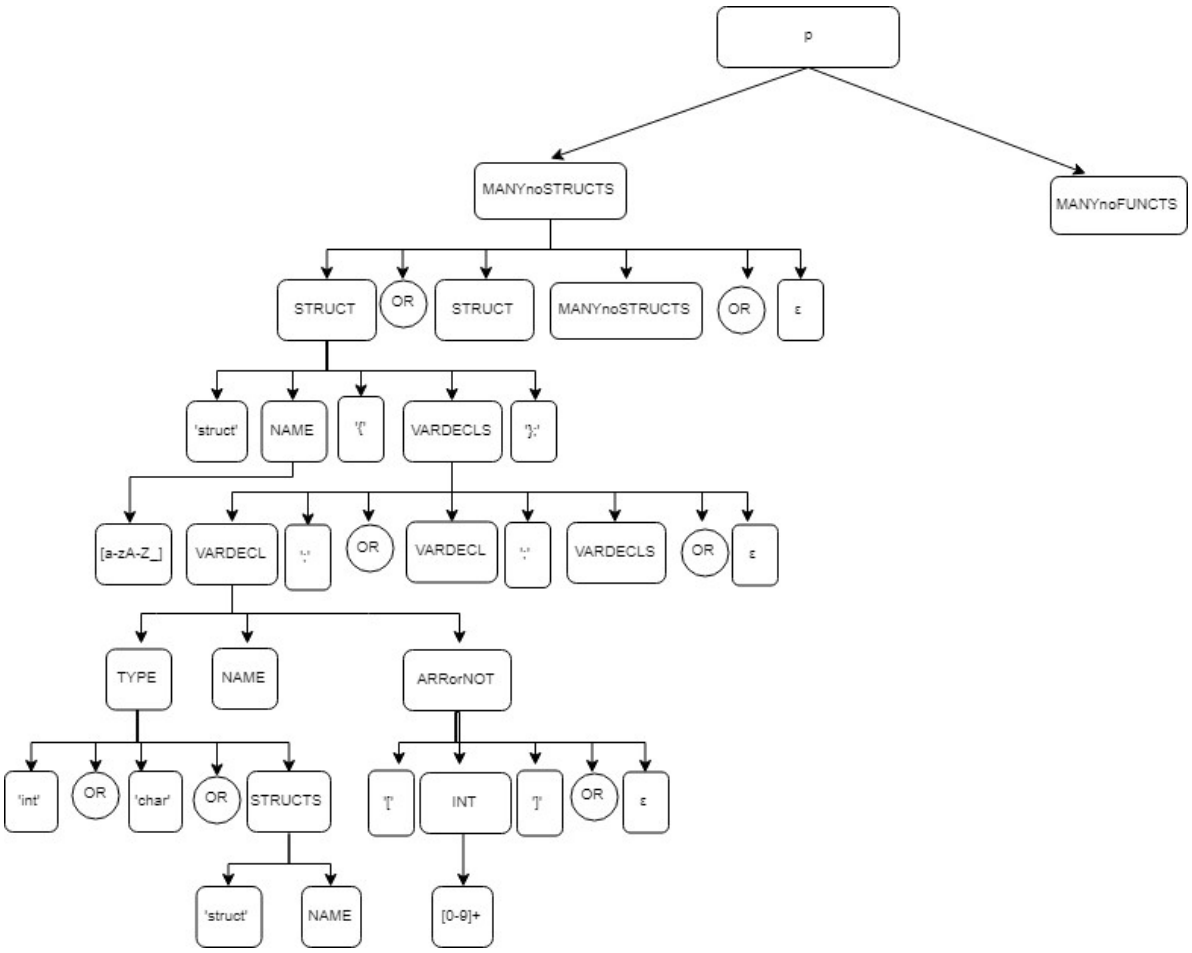
OP → AOP|BOP|COP

SEMIorNOT → ';'|ε

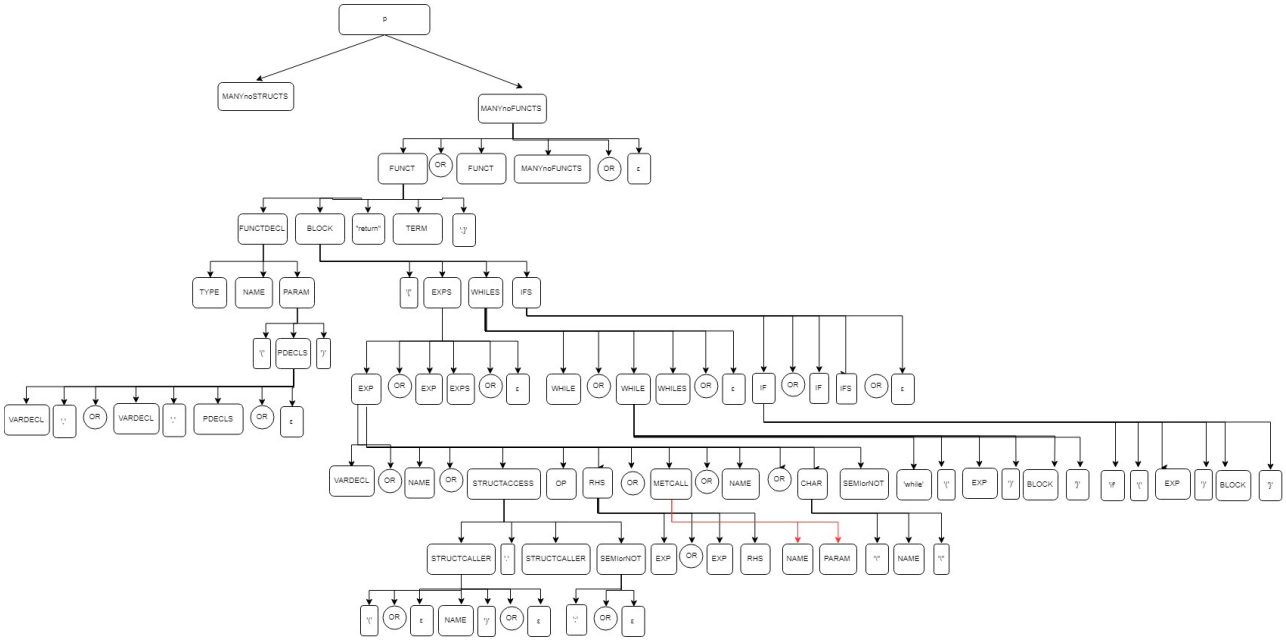
2.2 Ambiguity

An ambiguous grammar is a CFG for that there exists a string that can have more than one leftmost derivation or parse tree. An unambiguous grammar is a CFG for which every valid string has a unique leftmost derivation or parse tree.
To show that the above grammar is not ambiguous, the following tree has been created. It is visible that every valid string has a unique leftmost derivation

*Part 1: STRUCTS**



*Part 2: FUNCT**

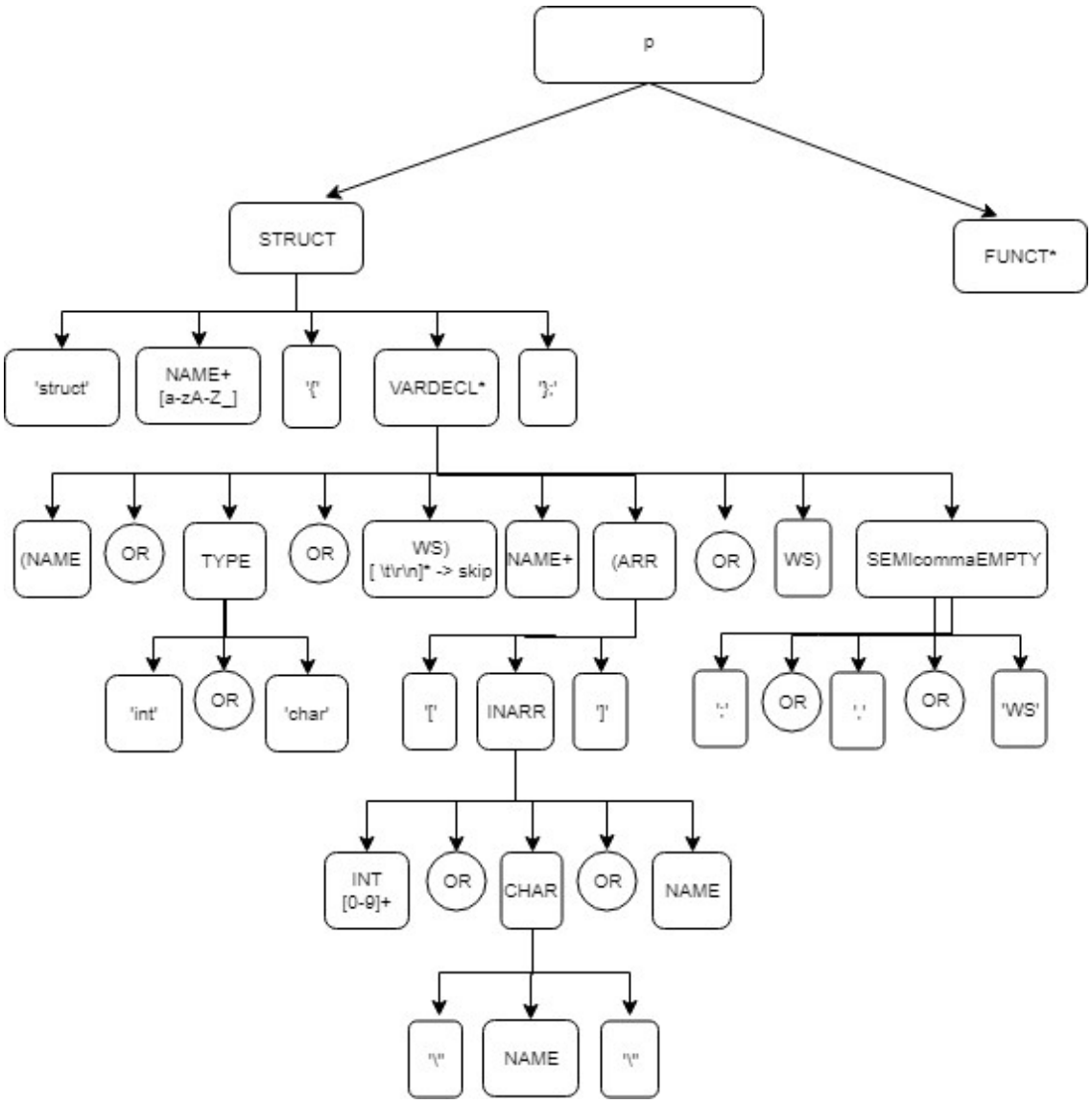


2.4 ANTLR rules implemented and tested on the 2.5 the code

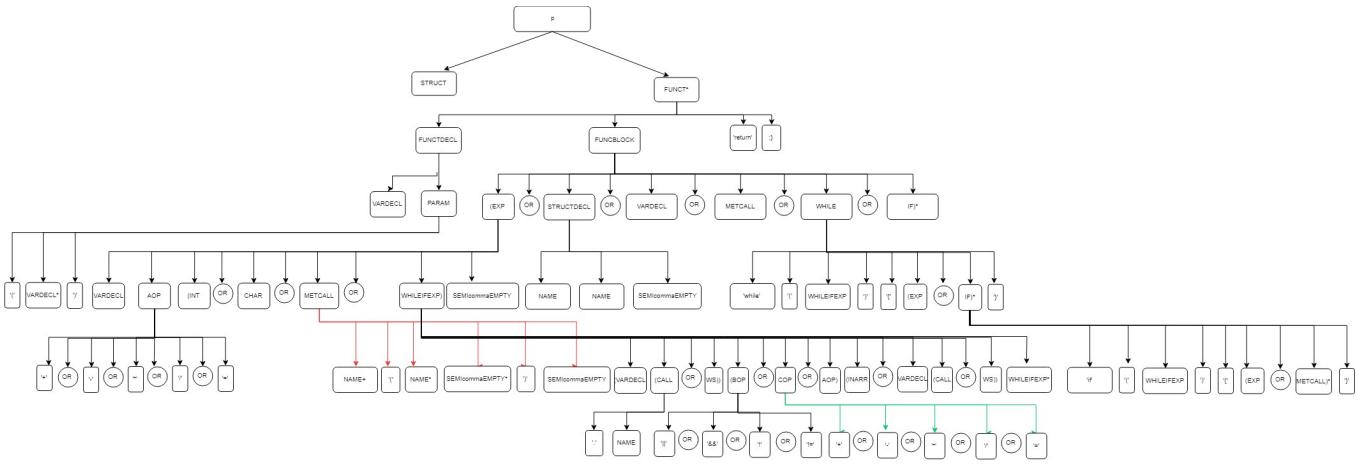
The implemented rules can be found in the submitted Hello.g4 file.

As a comparison to the above unambiguous tree, I have created a tree from the ANTLR rules as well. This has been done in order to compare as a sort of second “proof” since ANTLR does not support ambiguity.

*Part 1: STRUCTS**



*Part 2: FUNCT**


P
STRUCT
FUNCT*
FUNCTDECL
FUNCBLOCK
'return'
';'
VARDECL
PARAM
(EXP
OR
STRUCTDECL
OR
VARDECL
OR
METCALL
OR
WHILE
OR
IF)*
'('
VARDECL*
')'
VARDECL
AOP
(INT
OR
CHAR
OR
METCALL
OR
WHILEIFEXP(
SEMIcommaEMPTY
NAME
NAME
SEMIcommaEMPTY
'while'
'('
WHILEIFEXP
')'
'{'
(EXP
OR
IF)*
'}'
'+'
OR
'-'
OR
'*'
OR
'/'
OR
'%'
NAME*
')'
NAME*
SEMIcommaEMPTY*
'}'
SEMIcommaEMPTY
VARDECL
(CALL
OR
WS))
(BOP
OR
COP
OR
AOP)
((NARR
OR
VARDECL
(CALL
OR
WS))
WHILEIFEXP*
'if'
'('
WHILEIFEXP
')'
'{'
(EXP
OR
METCALL)*
'}'
'.'
NAME
'['
OR
'&&'
OR
'|'
OR
'!='
'<'
OR
'>'
OR
'='
OR
'!'
OR
'<='