Question that is always asked in the exam:

* Give you a problem, a feature of a language, an expression; then you do a CFG (context-free grammar) for this one and show it is ambiguous or not ambiguous.
* Enhance that with an attribute grammar and show the parse tree of an example.

**CFG FOR POSTFIX EXPRESSIONS**

* Suppose language only has integer literals and variables.
* Assume variables are already tokenized.
* Tokenization happens as part of the lexer. When we talk about CFG for an expression, we are talking about concrete syntax and maybe also abstract syntax.

Literals: integer

Variable: id 🡪 token comes from lexer

x 20 + 5 - 🡪 postfix expression (x + 20 – 5 🡪 infix notation)

Tokenization takes expression and returns this 🡪 id intv op+ intv op- (intv: integer value)

Now I can start writing my CFG.

I need a CFG that can define a postpix expression --------> for that we learned one notation: BNF

BNF needs to have 4 things:

* terminals (id, intv, op+, op-)
  + I am assuming my terminals are small cap letters
* non-terminals (i don’t know yet)
  + I am assuming my non-terminals are capital letters
* start non-terminal (will be one of my non-terminals)
* production rules

Start with defining the rules.

Define a non-terminal symbol 🡪 E (expression) : this is my start symbol

First production rule I want to write, rule to capture concrete syntax:

* E 🡪 E V OP | V V OP
  + I need to have non-recursive rule otherwise this will be infinite
  + I have non-terminal OP so I have to have a production rule for that one as well
  + V is value, it is non-terminal, I have to have a production rule for it as well
  + Operators are binary operators so they would require 2 values in front of them to be able to use
* V 🡪 id | intv
* OP 🡪 op+ | op-

E 🡪 E V OP | V V OP |

* You can do this ( for empty) if you have also unary operator.
  + V empty OP

Grammar supposed to have “E 🡪 a V”

* a should not have any non-terminals in it
* it could be “V a” : Left regular (a V is right regular)
* So for short, you can either have 1 non-terminal either on left or right

Then grammar is regular grammar

Our grammar is not regular grammar.

Our grammar:

* E 🡪 E V OP | V V OP
* V 🡪 id | intv
* OP 🡪 op+ | op-

Example:

lexer

* x 20 + 5 – --------------------> id intv op+ intv op-
* Can my grammar explain the output of lexer? Can I build a parse tree that can generate this output?

Parse tree starts with start symbol and at the leaves, it should have output of lexer in order.

E

E V OP

V V O intv op-

id intv op+

Is this an ambiguous (If you are able to generate two different parse trees by leftmost and rightmost derivation, then you have an ambiguous grammar.) grammar?

There is no algorithm to know if grammar is ambiguous or not.

If you are sure it is not an ambiguous grammar which will be the case if you have a regular grammar most of the time.

Can I go from concrete syntax to more abstract syntax?

What we have done is concrete syntax. If I can find a parse tree for (id int + int -), then it is instance of my language as far as the concrete syntax is concerned.

I need to look what it means, what is the abstract semantic, what does it mean by “id int +” (which means here the value). So in that case we use attribute grammars.

Attributed grammar 🡪 start with your CFG and add attributes to it.

What are the attributes that I care about meaning what is that semantic that I want to generate? When I say expression, semantic is known. What is the abstract semantic for expression? 🡪 VALUE. Expressions are values. If it was a statement, you will say what is the side effect is going to be.

When we talk about value, we consider 2 things: value that expression generates and type of the expression. We need these 2 things to define the complete semantic of the expression.

I need 2 attributes, one is value and other is type.

* Value
  + I am assuming that every terminal and non-terminal symbol (E, V, …) needs to have this value attribute.
  + Can I generate values from the production rules?
  + What is the value of x? x is a variable. We have a lot of attributes for the variables or features like where is it, when is this seeing the scope of it, etc. So there has to be some semantic regarding the variable itself. Assume that that semantic is handled so variable will have a value. So this value will come from my variable system whoever is managing that. So for my non-terminal symbols in this particular grammar will assume that the attributes for those terminal symbols will come from someone.
  + intv comes from tokenizer. Tokenizer says I found a literal here and this literal is following this format and this format means an integer number.
  + There is nothing defined for operator. You cannot say operator has a value. Operator is gonna do something on 2 values to generate a new value.

I go bottom up to find values.

Synthesized attributes:

* You take a production rule, take everything from the right hand side and you can calculate the left hand side’s value or attribute.

E

E V OP

V V O intv op-

id intv op+

Here, intv value is 20, other (right) intv is 5.

Let’s say id’s value is -5. Where did it come from? from my other attribute grammar whatever that is.

* E 🡪 E V OP | V V OP
  + E.value = V.value *(operator)* V.value
  + E.value = E.value *(operator)* V.value
* V 🡪 id | intv
  + V.value = intv.value
  + V.value = id.value
* OP 🡪 op+ | op-

op.f(\_, \_) 🡪 operator function takes the first and second value. This is gonna be defined by lexer.

Typing is top down.

Inherited attributes: typing could be one of these attributes.

E 🡪 V V O

* If first V is int, then second one has to be int. Because operator wants them to be same. This is one rule.
* Operator may say I always want integers. So in order to find type of V and other V, I need to know what the operator is thinking. If type is wrong, there is a syntax error.
  + Inherited thing happens from sibling.

For “type var;” kind of thing, type comes from the parent and push down.