- 1:

- Hello, my name is Artem,
- I am a student in JetBrains Programming Languages and Tools Lab.
- I am gonna tell you about using ECFG – the most readable form of programming languages syntax specification//

  in parsing with Generalized LL Algorithm

- ORACLE

- To specify syntax for a programming languages ,
- people usually use context-free grammars.
- In actual documentation grammars contain such constructions as
- one or more, zero or more, grouping and so on.
- Their primary **goal** is to increase readability and also decrease grammar size.
- The most common format to write grammar specification is
- **Extended Bachus Naur Form** which is just a standardized way to express **Extended Context-Free grammars.**

- ECFG

- This form allows a right-hand side of production to be a
- regular expression over the alphabet of terminals and noterminal
- Unfortunately, most widely-used parser generators can not process
- grammars in Extended Context Free form without transformation
- to a standard Context-Free form
- where right-hand side of productions can **only contain top-level alternatives**.
- Such transformation can highly increase the size of the grammar.

- Transformation

- For example, this little piece of java language grammar after conversion
- contains 18 nonterminals while it only used 7 nonterminals before.
- In parsing, size of a grammar matters.
- Parsers are, in general, sensitive to grammar size.

- Oracle again

- What is even worse, grammars which are used in language specifications

Click

- often are not LL so actual parser generator tools cannot process them.
- Developers have to convert grammar to the proper class for parser usage.

- Existing solutions

- Of course, the ton of work on parsing have been done already.
- There are tools like Yacc or Bison which are widely used for parsers development,

click

- but they do not admit grammars in the most natural form,
- and they can process just subclasses of context-free languages.
- Click
- There are also a bunch of research on ECFG processing without any prior transformations,
- click
- but, first of all, there are no tools
- And approaches, again, only can parse subclasses of context-free languages. --------click

- On the other hand,
- there exists a family of different Generalized parsing algorithms,
- for example, Generalized LL,
- Click
- which admit arbitrary CFG in a polynomial time,
- but none of them can process Extended CFG without first transforming it.
- Click

- So Our research is aimed to
- use Generalized LL algorithm for ambiguous ECFG grammars parsing.

- RA

- Sooo.... Let’s proceed to our work itself, shall we?
- In order to explain, how we what to achieve the goal
- I will first introduce some helpful concepts.

- First one is a suitable representation of a grammar.
- As a right-hand size of a production is a regular expression
- it can be represented as a special kind of automaton.
- Such automata are called Recursive automata and in the context of our work the only difference between the familiar to everyone Finite State Automata and RA is that the transitions — the edges at the pictures can be labeled not only with terminals, but also with nonterminals of the grammar EXPLAIN

- MINIMIZATION

- After construction of Recursive Automaton for grammar,
- we can easily minimize the resulting automaton with any minimization algorithm for Finite State Automata.
- It decreases the size of representation.

- explain. Long k-chains. Which can be merged. Note that this two components merged because
- for example this and this states are equivalent

- So we will use such data structure in the parsing process instead of usual grammar

- TREES
- So what means// to derive a string in this grammar representation?
- This means to find a path in the automaton from the initial state to any final,
- but every time we come across nonterminal transition, before going into the following state,
- we should first find some path from the state corresponding to that nonterminal to any final state, and only then transit.
- Doing this we naturally construct a derivation tree for a given string.
- So derivation tree is a rooted tree with a start nonterminal as a root, only leaves are terminal,
- and each internal node which correspond to a nonterminal has a sequence of children such that there exists a corresponding path in the automaton. EXPLAIN THEN go to next text

- Because of ambiguities in grammar, there exist 3 derivation trees
- But they share a lot of data between each other
- And it makes it rather inefficient to construct and store all of them just as they are.

- SPPF

- Fortunately, handy data structure Shared Packed Parse Forest can significantly reduce space usage.
- This data structure is well known in the Generalized Parsers family, and it is constructed by our favorite GLL
- so we use it.
- In the picture you can see an example of the forest packed from the set of trees from the previous slide.
- These rectangular nodes should not confuse you, they are only used to provide better sharing.
- All of them are here. (click-click-click).

- descriptors

- So, now, after we introduced necessary notions we can proceed to our modification of GLL algorithm.
- The basic GLL algorithm can be described as a process of juggling around with so called descriptors.
- Descriptor is a tuple which uniquely describes the whole parsing state.
- It consist of a position in grammar and input,
- the node in the parsing stack
- And Current forest root
- The algorithm basically creates descriptors,
- puts them into the q,
- and for each dequeued descriptor, creates the new set of descriptors and it goes on and on until the q becomes empty. Click---

- We are working with the automaton representation of the grammar
- so the grammar position is substituted with a state of the automaton.
- This is the main modification of the descriptor structure.

- Now I will show you how the descriptor handling process has been modified.
- I will demonstrate you this on an example to not tire you with formalities.
- First, let’s look at how the parsing goes when we deal with a grammar.
- We will use this grammar as an example.
- It can be transformed (click) to this grammar with the new nonterminal
- Now, when the grammar in the proper form, we can start deriving the string bc.
- We start moving in the grammar, creating and queueing descriptors for every starting position.(3 clicks)
- Then(click)we dequeue the first descriptor
- and try to move in input and grammar simultaneously.
- For this position we cannot move any further, so we dequeue the next descriptor.
- (click)For this descriptor we create the terminal node
- and make a step in grammar and input.
- Now we are standing before a nonterminal, so we create and enqueue descriptors for this positions(2click) ……

- I skip some similar steps..
- and for this position we build terminal node and reach end of the production
- So we also need to build nonterminal node
- So then the process goes on and on until we have no descriptors left in the q

- Automata
- This was a short explanation of how basic GLL algorithm works.
- When we deal with Recursive Automaton, some things become more complicated.
- So, at first, we only need to create a single descriptor — instead of three
- Only for initial automaton state
- But we need to handle all 3 transitions now.
- click
- For example for this terminal transition
- we should construct terminal node
- But the next state is final, so we also need to construct a nonterminal node.
- And both cases lead to deferent parsing states.

- Eval

- We have implemented both: basic GLL algorithm and our modification
- and compared them on this highly ambiguous grammar.
- Modified algorithm demonstrates
- 30 percent decrease in space usage
- and tooks 40 percent less time.

- These results may seem not that significant to you CLICK

- but in the context of graph parsing it makes a difference
- especially in space consumption.

- Graph parsing is when we have multiple input strings
- And we want to parse them all simultaneously
- So we represent them as a paths in single graph
- And just parse it.
- So there are some graph parsing results.
- As u can see memory usage decreased by 60 percent
- And time decreased by 45 percent.
- ----------NEXT->

- Graph parsing can be useful in a range of different areas
- such as string-embedded code analysis
- or graph databases querying
- or even in bioinforma’tics.
- This is our current focus: we are currently working on modification of presented GLL for graph parsing.
- Our preliminary experiments show practicability of such modification.

- PAUSE

- Thank you for your attention!
- If you have any questions I will be glad to answer them.