step 1. Read the "./generated/Python3.g4" file, line 110 to 145;

(this requires you to understand the grammar of regular expressions)

In the following example, we assume the grammar is:

```
    plus: NUMBER '+' NUMBER;
    NUMBER:[0-9]+;
    ADD:'+';
```

step 2. Find info from the "./generated/Python3Parser.h" file.

For example, you can find the following code with respect to the example grammar above (but not the actual grammar of the homework)

```
//..............
    class PlusContext : public antlr4::ParserRuleContext {
    public:
        PlusContext(antlr4::ParserRuleContext *parent, size_t invokingState);
        virtual size_t getRuleIndex() const override;
        std::vector<antlr4::tree::TerminalNode *> NUMBER();
        antlr4::tree::TerminalNode* NUMBER(size_t i);
        antlr4::tree::TerminalNode* ADD();

        virtual void enterRule(antlr4::tree::ParseTreeListener *listener)
override;
        virtual void exitRule(antlr4::tree::ParseTreeListener *listener)
override;

        virtual antlrcpp::Any accept(antlr4::tree::ParseTreeVisitor *visitor)
override;
    }
//..............
```

In the plus grammar the "NUMBER" occurs more than once, so there are two function in PlusContext correspondingly:

```
    std::vector<antlr4::tree::TerminalNode *> NUMBER();
    antlr4::tree::TerminalNode* NUMBER(size_t i);
```

The first returns the vector of pointers pointing to all NUMBER elements, the second returns the No.i NUMBER element (notice that the first one is No.0 instead of No.1).

Nevertheless, in the plus grammar the "ADD" occurs only once, so there is only one function in PlusContext about it:

```
    antlr4::tree::TerminalNode* ADD()
```

Which returns the pointer to the only ADD element.

For a terminal node, there are methods like:

```
//..............
std::string toString()
Token* TerminalNodeImpl::getSymbol()
/*
 * for example, consider:
 * antlr4::tree::TerminalNode *it;
 * it->toString() returns the string, for example, "123456" or "a"
 * (so you need to converse (std::string)"123456" to (int)123456)
 * it->getSymbol()->getTokenIndex() returns where this word is in the whole
input.
 */
//..............
```

You can read more example in part "notice" below.

step 3. Complete the "./src/Evalvisitor.h" by adding codes like:

```
//..............
    antlrcpp::Any visitPlus(Python3Parser::PlusContext *ctx)
    {
        /*
         * TODO
         * the pseudo-code is:
         * return visit(ctx->NUMBER(0))+visit(ctx->NUMBER(1));
         */
    }
//..............
```

When you write the

```
    visit(ctx->NUMBER(0))
```

the program will actually run code:

```
    visitAtom(ctx->NUMBER(0))
```

so you need and only need to use "visit" instead of "visitBalabala". (Think why this make sense?)

step 4. Compile the program. You only need to use the following commad:

```
cmake .
make
```

It is obvious that you need to install cmake on your Linux.

notice: the Antlrcpp::Any.

This is a class and you now only need to know two things about this: the is() method and as() method.

For example, if the grammar is:

```
plus: atom '+' atom;
atom: NUMBER | STRING+;
NUMBER:[0-9]+;
STRING:[A-Z]+;
ADD:'+';
```

The Context in Parser.h is like:

```
//..............
    class PlusContext : public antlr4::ParserRuleContext {
    public:
        PlusContext(antlr4::ParserRuleContext *parent, size_t invokingState);
        virtual size_t getRuleIndex() const override;
        std::vector<AtomContext*> atom();
        AtomContext* atom(size_t i);
        antlr4::tree::TerminalNode* ADD()

        virtual void enterRule(antlr4::tree::ParseTreeListener *listener)
override;
        virtual void exitRule(antlr4::tree::ParseTreeListener *listener)
override;

        virtual antlrcpp::Any accept(antlr4::tree::ParseTreeVisitor *visitor)
override;
    }
    //notice that the atom is AtomContext* type rather than a TerminalNode* type.
    //an easy way to tell the difference is: capital letter-TerminalNode;
xxxContext otherwise
//..............
```

A part of the correct code may be like:

```
//..............
    antlrcpp::Any visitPlus(Python3Parser::PlusContext *ctx)
    {
        antlrcpp::Any ret1, ret2;
        ret1 = visit(ctx->NUMBER());
        ret2 = visit(ctx->NUMBER());
        if (ret1.is<int>() && ret2.is<int>())
            return ret1.as<int>() + ret2.as<int>();
        else if (ret1.is<std::string>() && ret2.as<std::string>())
            return ret1.as<std::string>() && ret2.as<std::string>();
        else if (ret1.is<int>() && ret2.is<std::string>())              //no
need
            throw("unsupported operand type(s) for +: 'int' and 'str'");//no
need
    }
//..............
```

We prove that the test files are correct, so the final two lines are not needed.

You are suggested to use "is()" before an "as()".

However, during the OOP course, you will learn the constructing function and destruction function. You'd better to understand how the Any constructs and destructs, which will help you debug. You can read "./third_party/runtime/src/support/Any.h" and Any.cpp in the same path to understand it. This may be hard, so we suggest you ask teaching assistant JHY for help.

Optional requirement: read the following context to know how antlrcpp::Any works:

https://www.cnblogs.com/mangoyuan/p/6446046.html

https://www.cnblogs.com/xiaoshiwang/p/9590029.html

You can search "c++11 traits" to find more documents.