BASIC

Brief introduction of this project

This project is a mini version of BASIC made by Microsoft. It is an interpreter which provides us with serveral command and statements to make another program with the `BASIC` language.

Compared with many High-level languages like `C++` and `Java`, `BASIC` is a relatively original language. But in this project, mini BASIC is even more simple than BASIC made by Microsoft.

Description of class

- **Widget :** Class `Widget` is declared in `widget.h`. It is used to hold the consolo widget.
- **Consolo:** Class `Console` is declared in `console.h`. Class `Console` is used to hold the input from user. It parses the command and dispatch the command to different member functions of class `Console`.
- **Buffer:** Class `Buffer` is declared in `buffer.h`. Class `Buffer` is used to store what user has input to the interpreter.
- **Program** Class `Program` is declared in `program.h`. Class `Program` is used to run the code input by user.
- **Tokenizer** Class `Tokenizer` is declared in `tokenizer.h`. Class `Tokenizer` is used to tokenize each line of code and divide them into different parts which is good for the subsequent parsing.
- **EvalState:** Class `EvalState` is declared in `evalstate.h` which is used to store the variables and their value.
- **Parser:** Class `Parser` is declared in `parser.h` which is responsible to parse the expressions and calculate them. Checking the cerrectness of code is one of its functions.
- **Statement:** Class `Statement` is declared in `statement.h`. Class `Statement` has ten subclasses, which corresponse to ten different kinds of statements.
- **Expression:** Class `Expression` is declared in `expression.h`. Class `Expression` has three sub classed, which corresponse to class `ConstantExp`, class `IdentifierExp` and class `CompondExp`.

How the program works

I will introduce the structure from top to bottom.

First, the whole widget is a big input box where user can type anything. The input box corresponses to class `Console`. `Consolo` is set in a blank widget which is generated by class `Widget`. When user finishes inputing command and press 'Enter', the command will be sent as a signal form function `Console::keypressEvent()` to function `Console::dispatchCmd()`. Then `Console::dispatchCmd()` will parse the command and dispatch it to corresponding member

function in class `Console`. If what user inputs is code, `Console` will pass the code to `Buffer` which can store lines of code using linked list and sort them by line number from small to large. When user finishes inputting code and then inputs the command of `RUN`, `Console` will instantiate the class `Program` which starts to parser the code stored in `Buffer` and run these lines of code according different kinds of statements.

Next, before run lines of code, some preparation should be made. Class `Tokenizer` is stored in a array named `Tokenizer ** Program::code` as pointer. Each `Tokenizer` corresponds to a line of code. `Tokenizer` divides code into line number, statement and remanent string named `QString Tokenizer::others`. `QString Tokenizer::others` will be further processed by function `Tokenizer::furtherToken()` to remove needless space. Besides, each `Tokenizer` contains a pointer of class `Expression` and a pointer of class `Statement`. In class `Program`, there is a member variable named `int Program::pointer` which is similar to `%rip` in assemble code and tell `Program` which code should be excute next. Then `Program` can run code form line to line according to the value of `int Program::pointer`.

Then I will introduce how the class `Statemnet` and class `Parser` works.

Statement

- **RemStatement:** `RemStatement` does nothing so the program will jump over it.
- **LetStatement:** `LetStatement` calculate the expression in `QString Tokenizer::others` and assign the value of result to identifier. The specific process of caluculation is provided in Parser.
- **InputStatement:** `InputStatement` receive numbers from user and assign the value to identifier. It changes the value of a flag named `bool Program::hang` which tell `Program` whether program is in input mode. In this mode, `Program` stops execute code and wait for user to enter a number.
- **PrintStatement:** `PrintStatement` print the result of expression in `QString Tokenizer::others`. The specific process of caluculation is provided in Parser.
- **GotoStatement:** `GotoStatement` change `int Program::pointer`, making program jump to specific line of code.
- **IfStatement:** `IfStatement` first analysises `QString Tokenizer::others` and divides it into two parts, logic condition and target address. Calculation of logic condition is left to `Parser`. Jumping part is similar to `GotoStatement`.
- **EndStatement:** `EndStatement` change the value of a flag named `bool Programm::end` which tell `Program` to stop executing the code.

Parser

Class `Parser` is responsible to build expression tree, calculate the expression and check the gramma. Expression in `QString Tokenizer::others` is build in binary tree based on class `Expression`. For calculation, `Parser` uses recursive method to traverse the tree.

Advance function

Besides the basic function of mini BASIC, the ability to call function is added to the project. To do this, three new statement is added to class `Statement`.

- **SubStatement:** `SubStatement` add function name and corresponding line number to `map<QString,int> Program::funList`. Then change `int Program::pointer` behind next `End`.
- **EndSubStatment:** `EndSubStatment` pop line number from `stack<int> Program::return_addr` and jump to the line number.
- **CallStatement:** `CallStatement` search function name in `map<QString,int> Program::funList`, push the next line number to `stack<int> Program::return_addr` and jump to line number where the function is.

Something need to improve

1. Process of exception should be more specific. In the current stage, try and catch can hold most exception, pointing out where the erro happend. But it can't tell the specified place where the erro happened.
2. `LET` and `INPUT` can't be execute directely without typing line number. Because I don't konw what namespace in which these two commands will affect variables. If all the program share the same namespace, will execution of program be affected by that executed previously? I don't think so. Besides, it's not hard to realize the function, given that I have already realized the `PPRINT` command without typing line number.