School of Computing | Chansu Park
September 27, 2018 | 20173245

CS420
Fall 2018, Assignment #1

1 Policies of this project

1.1 How to make and execute

This coursework has been built on `g++ 7.3.0` with `Ubuntu 18.04`. The language is `C++`.
There are four source codes: `arith-ast.cpp`, `arith-scanner.cpp`, `arith-parser.cpp` and `arith-main.cpp` in the folder `src`.
You can find `Makefile` in the root. The command `make` or `make arith-main` compiles all source code and generate an executable `arith-main` in the folder `bin` which takes `input.txt` and prints the result into `output.txt`. Since the input name was fixed as `input.txt`, the implementation only accepts an input file names as `input.txt`. Note that input.txt should be located in the folder that you execute `arith-main`. (For instance, if you execute at the root folder, the command will be `./bin/arith-main`, where `input.txt` should be located in the root.) I located the sample `input.txt` and `output.txt` in the `bin` folder.
Finally, you can find this report in the root folder.

1.2 Assumed points

- I assumed that the empty line itself wouldn't be an input.
- I followed left-associative form.
- I filtered only digits and alphabets (both in UPPERCASE and lowercase) as valid characters.
- I don't allow numbers starting with 0s.
- For each input line, I take them as a `stringstream` object.

2 Implementation

2.1 Scanner

Scanner needs toknization. I defined tokens as a class `Token` in the scanner:

Data Types	`tNumber, tID`
Arithmetic Operators	`tPlusMinus, tMulDiv`
Special Tokens	`tStart, tEOF, tError, tUndefined`

First two rows are obvious. For the third row:

`tStart`: The dummy token that alerts scanner to initialize new scan. Thus the scanner has a `tStart` token when it is initialized.

`tEOF`: When the given `stringstream` input meets `EOF`, put `tEOF`.

`tError`: Whenever it meets invalid character, token becomes `tError` since this error is scanning error.

`tUndefined`: When initializing an empty token, it becomes `tUndefined` token.

Note that, since our aim is constructing LL(1) parser, the scanner saves only one token at a time. Whenever scanner detect new token, it serves that token to the parser if needed and read new token.

There are several methods in the class `Scanner`. I took a note for some important methods for this scanner.

`scan()`: This method scans new valid token. It first ignores all whitespaces. When it meets one of the arithmetic operators, it saves a new token either `tPlusMinus` or `tMulDiv`. When it meets alphabet, it saves a new identifier token `tID` with its name. When it meets digit, it peeks and reads next character until it faces non-digit character, and save it as a string in the `tNumber` token.

`nextToken()`: It deletes current token and perform `scan()` to get a new token.

`getNext()`: It copies current token, call `nextToken()` to perform a new scan, and return the copied token.

`peekNext()`: It checks current token and returns.

2.2 Abstract Syntax Tree

The input expression can be realized as an arithmetic tree. For now, every node is an implementation of virtual class `AstExpression`. `AstExpression` node can be realized on of two types of nodes where the second type of nodes can be derived into two different types.

1. Binary operation nodes (`AstBinaryOp`): It has one of the four arithmetic operators as a value and has two childs.
2. Operand nodes (`AstOperand`): It is a leaf node and has a string itself as a value.
   - 2.1 Identifier nodes (`AstIdent`): Identifier node. It contains the name of the identifier.
   - 2.2 Constant nodes (`AstConstant`): Constant node. Currently only numbers (multi-digit) will be accepted in this form.

Note that any node contains the corresponding token.

2.3 Parser

Parser peeks the next token of scanner to decide whether or not that token has a type where the current rule needs. Then it reads that token and let scanner to prepare the next token. You can see the phenomena in the `nextToken(...)` function of the parser.
Also there are functions that correspond to each terminal/non-terminal symbol.

`goal()`: It has a return type `AstNode*`. Since it is a starting symbol, it first reads `tStart` token and call `expression()`. If `expression()` returns `NULL` or the next token is not `tEOF`, it generates an error.

`expression()`: It has a return type `AstExpression*`. Call `term()`, check an error, and call `expressionP()`. Only if `expressionP()` returns a partial `AstBinaryOp` pointer, it attaches the result of `term()` as a left child of that pointer and return. Otherwise it returns the result of `term()` itself.

`expressionP()`: It has a return type `AstBinaryOp*`. If "+" or "-" is detected, it reads that token, call `expression()` and check whether or not it is empty, since then the right operand is needed. Then it generates the node with empty left. If it fails to detect "+" or "-" at the beginning, it returns `NULL` without error.

`term()`: Similar to `expression()`: change corresponding `term()` to `factor()`; `expressionP()` to `termP()`.

`termP()`: Similar to `expressionP()`: change corresponding "+" or "-" to "*" or "/"; `expression()` to `term()`.

`factor()`: It has a return type `AstOperand*`. If the next token is either `tIdent` or `tNumber`, return the corresponding token with value using `ident()` and `constant()`.

3 Sample input and output

Here are some valid/invalid inputs and corresponding outputs in the left-associative form:

Input	Output
`x-2*y`	`-x*2y`
`a + 35 - b`	`+a-35b` (Ignores whitespaces, left-associative)
`10+*5`	incorrect syntax ("*" after "+")
`10 20`, `a 2`, `2 a`, `a b`	incorrect syntax (two consequent operands without operators)
`2a`, `a2`	incorrect syntax (ID and digits are attached without spaces)
`ab`	incorrect syntax (ID can have only 1 character)
`1%2`	incorrect syntax (invalid character detected)
`035`	incorrect syntax (number starting with 0)