Упражнение 4

В этом задании будет разработан лексический анализатор языка MiniC. Задание следует решать в группах по 2-3 человека, используя Git для совместной работы над кодом (смотри инструкцию http://prog.tversu.ru/pr3/Git&MSVC.pdf).

В лексическом анализаторе для представления категории лексемы будет использовано следующее перечисление:

```
enum class LexemType { num, chr, str, id, lpar, rpar, lbrace, rbrace, lbracket, rbracket,
semicolon, comma, colon, opassign, opplus, opminus, opmult, opinc, opeq, opne, oplt,
opgt, ople, opnot, opor, opand, kwint, kwchar, kwif, kwelse, kwswitch, kwcase, kwwhile,
kwfor, kwreturn, kwin, kwout, eof, error };
```

Тип `LexemType` содержит все типы лексем учебного языка MiniC плюс два вспомогательных типа мета-лексем: `eof` – для обозначения конца потока лексем и `error` – для обозначения лексической ошибки.

Для распределения кода по файлам можно использовать следующие стратегии:

Вариант 1	**Вариант 2**
Объявление всех классов и перечисления LexemType размещаются в файле LexicalAnalyser.h. Реализация всех классов помещается в файле Scaner.cpp. Функция main размещается в файле compiler.cpp. В файлах, в которых будет использоваться лексический анализатор (пока это тесты и compiler.cpp), делается #include "Scaner.h". Тесты может быть целесообразно разбить на несколько файлов, по одному файлу на тестируемый класс.	Для каждого класса делается два файла - .cpp и .h. В заголовочные файлы помещаем объявление соответствующего класса, определение его функций – в файл .cpp. Перечисление LexemType размещается в файле Token.h (где объявлен класс Token). В файле Scaner.h (с объявлением лексического анализатора) подключается файл Token.h. Функция main размещается в файле compiler.cpp. Из этого файла подключается Scaner.h. Так как тестов может быть достаточно много, их можно разбить на несколько файлов, и в каждом подключать заголовок с тем классом, который тестируется.

Шаг 1

Реализуйте класс `Token` для токенов лексического анализатора. Полями класса являются:

- _type – тип лексемы (`LexemType`);
- _value – целое число, используемое следующим образом:
  - значение числовой константы для лексемы типа `LexemType::num`;
  - значение символьной константы (код символа) для лексемы типа `LexemType::chr`;
- _str – строка, в которую заносятся:
  - идентификатор, для лексемы типа `LexemType::id`;

- строка для лексемы типа `LexemType::str`;
- сообщение об ошибке для лексемы типа `LexemType::error`.

В классе должны быть реализованы следующие конструкторы:

- `Token(LexemType type)` – будет использоваться для лексем без параметров (`LexemType::lpar` и т.д.);
- `Token (int value)` – будет использоваться для лексем с целочисленным параметром (`LexemType::num`);
- `Token(LexemType type, const string & str)` – будет использоваться для лексем с значением-строкой (`LexemType::error`, `LexemType::id` и `LexemType::str`);
- `Token(char c)` – будет использоваться для лексемы типа `LexemType::chr`.

Конструкторы должны сохранить параметры в полях класса.

В классе должен быть реализован метод `void print(ostream &stream)`, который выводит описание лексемы в поток stream. Описание лексемы печатается в квадратных скобках, и включает имя лексемы (обязательно) и используемые в лексеме данного типа параметры, например:

```
[eof]
[id, "name"]
[chr, 'a']
[error, "символьная константа содержит более одного символа"]
```

Для перевода типа константы из `LexemType` в строку реализуйте дополнительный метод, использующий оператор switch.

Реализуйте методы для доступа к полям класса:

- `LexemType type();` – возвращает тип лексемы;
- `int value();` – возвращает целочисленное значение лексемы;
- `string str();`– возвращает строку лексемы.

Реализуйте тесты для класса `Token`. В тестах, в частности, следует проверить правильность вывода для **всех** типов лексем. Для тестирования используйте класс `ostringstream`.

Шаг 2

Реализуйте класс лексического анализатора `Scanner`.

Конструктор класса должен получать на вход ссылку на входной поток (`istream &stream`) и сохранять её в поле класса.

Основной метод класса лексического анализатора

```
Token getNextToken();
```

должен при каждом вызове возвращать следующий токен синтаксического анализа. Реализация функции должна основываться на схеме конечного автомата, построенного на лекции (схема автомата размещена на http://prog.tversu.ru/cs3/minicautomata.pdf).

При успешном завершении анализа по исчерпании входного потока возвращается токен [eof].

При возникновении лексической ошибки функция лексического анализа должна сохранять в поле value лексемы код ошибки – при этом нужно закодировать все возможные типы ошибок:

- неподдерживаемый языком символ (за исключением случаев, когда этот символ является частью строковой или символьной константы);
- отсутствие разделителя между символами операций;
- одиночный символ | или одиночный символ &;
- пустая символьная константа;
- символьная константа, содержащая более одного символа.

Для упрощения реализации процедуры getNextToken может быть полезно реализовать отображения (словари в терминах Python) для определения кода лексемы для знаков пунктуации и ключевых слов. Словари могут быть инициализированы следующим образом

```
#include <map>

map<char, LexemType> punctuation{ { '[', LexemType::lbracket }, { ']', LexemType::rbracket } };
map<string, LexemType> keywords{ { "return", LexemType::kwreturn } };
```

Для проверки наличия ключа в отображении можно использовать метод count(key), который возвращает число записей с таким ключом (и 0, если ключа нет). Для доступа к значению используется синтаксис с [ ], например `keywords["return"]`.

Реализуйте тесты для лексического анализатора. Подойдите к тестированию тщательно, так как ошибки в реализации лексического анализатора будут мешать реализовывать синтаксический анализатор. Для тестирования удобно использовать `istringstream`.

Разумный подход к тестированию лексического анализатора состоит в том, чтобы постепенно реализовывать тесты для различных переходов анализатора и добавлять их реализацию в метод `getNextToken()`. Начните с того, что убедитесь, что на пустом входе анализатор выдает токен [eof]. Затем добавьте тест на какую-нибудь одну лексему и реализуйте соответствующий переход автомата.

Убедитесь, что все переходы автомата протестированы как минимум один раз. Если в каких-то местах используется накопление информации во вспомогательных переменных (Digit, value на схеме автомата), протестируйте, что эти переменные корректно переинициализируются при повторном переходе в соответствующее состояние (так, чтобы предыдущее слово или число не оказалось «приклеено» к началу следующего).

Убедитесь, что везде, где происходит возврат из лексического анализатора, его внутреннее состояние устанавливается корректно (следующая лексема тоже будет правильно распознана).

Для тестирования удобно использовать короткие фрагменты из нескольких лексемами, но не забудьте добавить и несколько тестов с большим объемом анализируемого кода.

Шаг 3

Для проверки лексического анализатора реализуйте в функции main() код, который будет выполнять разбор заданного текстового файла и распечатывать полученные токены. Эта функция может выглядеть следующим образом:

```
int main(){
	ifstream ifile("myprog.minic");
	Scanner scanner(ifile);

	for (;;){
		Token currentLexem = scanner.getNextLexem();
```

```
            currentLexem.print(cout);

            if (currentLexem.type() == LexemType::error
                  || currentLexem.type() == LexemType::eof){
                  break;
            }
      }
}
```

Пример работы программы

Содержимое сканируемого файла myprog.minic	Вывод лексического анализатора

`char str[] = "Hello, world!";`
`int a = 5;`
`char c = 'c';`
`a = a + c;`
	`[kwchar]`
	`[id, "str"]`
	`[lbracket]`
	`[rbracket]`
	`[opassign]`
	`[str, "Hello, world!"]`
	`[semicolon]`
	`[kwint]`
	`[id, "a"]`
	`[opassign]`
	`[num, 5]`
	`[semicolon]`
	`[kwchar]`
	`[id, "c"]`
	`[opassign]`
	`[chr, 'c']`
	`[semicolon]`
	`[id, "a"]`
	`[opassing]`
	`[id, "a"]`
	`[opplus]`
	`[id, "c"]`
	`[semicolon]`
	`[eof]`

Оценка задания

Задание 2.1. Вывод лексем 80 %

Задание 2.2. Вывод лексических ошибок 20 %

Баллы за каждую задачу включают:

- правильность решения задания – 40 %;
- оформление кода согласно http://prog.tversu.ru/pr3/codeStyle.pdf – 10%;
- модульные тесты – 50%.

Задание является обязательным для получения зачёта в семестре.