Домашнее задание №1. Лексический анализатор для MiniC

В этом задании будет разработан лексический анализатор для языка MiniC на основе конечного автомата, построенного на лекциях (схема КА выложена в LMS).

В лексическом анализаторе для представления категории лексемы должно использоваться 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.

Используйте следующую стратегию для размещения кода по файлам:

- Объявление всех классов и перечисления `LexemType` размещаются в файле `Scaner.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.

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 со строковым значением (`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 в LMS).

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 лексем, 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	Вывод лексического анализатора

`char str[] = “Hello, world!”;`
`int a = 5;`
`char c = ‘c’;`
`a = a + c;`
	`[kwchar]`
	`[id, 0]`
	`[lbracket]`
	`[rbracket]`
	`[opassign]`
	`[str, 0]`
	`[semicolon]`
	`[kwint]`
	`[id, 1]`
	`[opassign]`
	`[num, 5]`
	`[semicolon]`
	`[kwchar]`
	`[id, 2]`
	`[opassign]`
	`[chr, ‘c’]`
	`[semicolon]`
	`[id, 1]`
	`[opassing]`
	`[id, 1]`
	`[opplus]`
	`[id, 2]`
	`[semicolon]`
	`[eof]`

Оценка задания

1. Вывод лексем	60 %
2. Вывод лексических ошибок	20 %
3. Наличие модульных тестов	20 %