МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Новосибирский государственный технический университет»

НГТУ НЭТИ

Кафедра прикладной математики

Практическая работа №2

по дисциплине «Языки программирования и методы трансляции»

ФПМИ

Группа	ПМ-92
Вариант	7
Студенты	Кутузов Иван
	Иванов Владислав
Преподаватель	Еланцева И. Л.
Дата	17.03.2022

Новосибирск

Цель работы:

Изучить методы лексического анализа. Получить представление о методах обработки лексических ошибок. Научиться проектировать сканер на основе ДКА

Задание:

Подмножество языка C++ включает:

- данные типа int, float, массивы из элементов указанных типов;
- инструкции описания переменных;
- операторы присваивания в любой последовательности;
- операции +, – , *, = =, != , <, > .

В соответствии с выбранным вариантом задания к лабораторным работам разработать и реализовать лексический анализатор. Исходными данными для сканера является программа на языке C++ и постоянные таблицы из предыдущей практической работы. Результатом работы является создание файла токенов, переменных таблиц и файла сообщений об ошибках.

Алгоритм:

Шаг 1.

Сначала разобьем исходный текст на лексемы по определенным правилам:

```
class LexemSplitter {
public:
	virtual vector<string> splitToLexemes(string text) = 0;
};
```

Собираем лексему (по одному символу из текста) пока она сопоставляется с шаблоном:

```
class Matcher {
public:
	virtual bool match(string text) = 0;
};
```

Разделим шаблоны на две категории: шаблон для записи и шаблон “пропускаемых” символов (например комментарии).
Класс реализующий разбиение исходного текста будет выглядеть следующим образом:

```
class BaseLexemSplitter : public LexemSplitter {
private:
	Matcher* _writtable;
	Matcher* _skippable;

public:
	BaseLexemSplitter(Matcher* writtable, Matcher* skippable);

	vector<string> splitToLexemes(string text) override;
};
```

реализация методов будет представлена ниже вместе с текстом программы

Шаг 2.

Теперь каждой лексеме, полученной в пункте 1 необходимо определить тип, который будет использоваться в синтаксическом анализе:

```
enum class Type {
	INTEGER_KEYWORD,
	FLOAT_KEYWORD,
	ARRAY_KEYWORD,
	INTEGER_CONSTANT,
	FLOAT_CONSTANT,
	ARRAY_CONSTANT,
	IDENTIFER,
	OPERATION,
	ASSIGMENT,
	END_OF_COMMAND,
	ERROR,
};
```

временно будем использовать enum

Лексема и ее тип вместо образуют структуру токена:

```
struct Token {
public:
	Type type;
	string key;

	Token(Type type, string key) {
		this->type = type;
		this->key = key;
	}
};
```

Для каждого типа опишем шаблон для сопоставления с лексемой:

```
class TypeDefiner {
public:
	virtual bool isType(string word) = 0;
};
```

Парсер лексемы в токен выглядит следующим образом:

```
class TokenParser : public Parser<Token> {
private:
	vector<pair<TypeDefiner*, Type>> _definers;

public:
	TokenParser(vector<pair<TypeDefiner*, Type>> definers);

	Token parse(string word) override;
};
```

Поскольку некоторые лексемы заранее известны и хранятся в таблице, то напишем класс заместитель, который сначала будет искать лексему в таблице:

```
class TokenParserWithTable : public Parser<Token> {
private:
	ReadOnlyTable<string, Type>* _table;
	Parser<Token>* _baseParser;

public:
	TokenParserWithTable(Parser<Token>* _baseParser,
ReadOnlyTable<string, Type>* table);

	Token parse(string word) override;
};
```

Теперь все готово для того, чтобы перевести исходный текст в список токенов:

```
class BaseTokenizer : public Parser<vector<Token>> {
private:
	Parser<Token>* _parser;
	LexemSplitter* _splitter;

public:
	BaseTokenizer(Parser<Token>* tokenParser, LexemSplitter*
lexemSplitter);

	vector<Token> parse(string text) override;
};
```

Тестирование

Тест	результат

`Parser<vector<Token>>* tokenizer = new BaseTokenizer(parser, splitter);`
 	
`string code = "int/*multi\nline\ncomment*/ a[ab]=0;\n//comment\n @% ==1.23;\n int[] array+({1, 2,3}) - {1, 2)";`
 	
`vector<Token> expected;`
`expected.push_back(Token(Type::INTEGER_KEYWORD, "int"));`
`expected.push_back(Token(Type::IDENTIFER, "a"));`
`expected.push_back(Token(Type::OPERATION, "["));`
`expected.push_back(Token(Type::IDENTIFER, "ab"));`
`expected.push_back(Token(Type::OPERATION, "]"));`
`expected.push_back(Token(Type::OPERATION, "="));`
`expected.push_back(Token(Type::INTEGER_CONSTANT, "0"));`
`expected.push_back(Token(Type::END_OF_COMMAND, ";"));`
`expected.push_back(Token(Type::ERROR, "@%"));`
`expected.push_back(Token(Type::OPERATION, "=="));`
`expected.push_back(Token(Type::FLOAT_CONSTANT, "1.23"));`
`expected.push_back(Token(Type::END_OF_COMMAND, ";"));`
`expected.push_back(Token(Type::INTEGER_KEYWORD, "int"));`
`expected.push_back(Token(Type::OPERATION, "["));`
`expected.push_back(Token(Type::OPERATION, "]"));`
`expected.push_back(Token(Type::IDENTIFER, "array"));`
`expected.push_back(Token(Type::OPERATION, "+"));`
`expected.push_back(Token(Type::OPERATION, "("));`
`expected.push_back(Token(Type::ARRAY_CONSTANT, "{1, 2,3}"));`
`expected.push_back(Token(Type::OPERATION, ")"));`
`expected.push_back(Token(Type::OPERATION, "+"));`
`expected.push_back(Token(Type::ERROR, "{1, 2)"));`
 	
`auto actual = tokenizer->parse(code);`
 	
`for (int i = 0; i < actual.size(); i++) {`
`if (actual[i].type != expected[i].type || actual[i].key != actual[i].key) {`
`return false;`
`}`
`}`
 	
`return true;`
	true

`Parser<vector<Token>>* tokenizer = new BaseTokenizer(parser, splitter);`
 	
`string code = "float b /* int a = 0; float b = 1; a = b - a; *";`
 	
`vector<Token> expected;`
`expected.push_back(Token(Type::FLOAT_KEYWORD, "float"));`
`expected.push_back(Token(Type::IDENTIFER, "b"));`
 	
`auto actual = tokenizer->parse(code);`
 	
`for (int i = 0; i < actual.size(); i++) {`
`if (actual[i].type != expected[i].type || actual[i].key != actual[i].key) {`
`return false;`
`}`
`}`
 	
`return true;`
	true

Текст программы?

Lexeme splitter

Matchers.h

```
class Matcher {
public:
	virtual bool match(string text) = 0;
};

class RegexMatcher : public Matcher {
private:
	regex _pattern;

public:
	RegexMatcher(regex pattern);

	bool match(string text) override;
};

RegexMatcher::RegexMatcher(regex pattern) {
	_pattern = pattern;
}

bool RegexMatcher::match(string text) {
	return regex_match(text, _pattern);
}

class OperationMatcher : public RegexMatcher {
public:
	OperationMatcher() : RegexMatcher(regex(R"([-+=*<>!\[\]\(\)]|={2}|!=)")) {};
};

class WhiteSpaceMatcher : public RegexMatcher {
public:
	WhiteSpaceMatcher() : RegexMatcher(regex(R"(\s+|\R+)")) {};
};

class EndOfCommandMatcher : public Matcher {
private:
	string _endOfLine = ";";

public:
	bool match(string text) override;
};

bool EndOfCommandMatcher::match(string text) {
	if (text == _endOfLine) {
		return true;
	}

	return false;
}

class OtherSymbolsChainMatcher : public Matcher {
private:
	regex _other = regex(R"([^\s-+=*<>\R!;/\[\]\(\)]+|^\{.*\}?)");

public:
	bool match(string text) override;
};

bool OtherSymbolsChainMatcher::match(string text) {
```

```
	if (regex_match(text, _other)) {
		if (text.front() == '{' && text.size() > 2) {
			if (text[text.size() - 2] == '}') {
				return false;
			}
		}

		return true;
	}

	return false;
}

class CommentMatcher : public Matcher {
private:
	regex _pattern = regex(R"((^//.*|^/\*(.|\n)*(\*/)?))");

public:
	bool match(string text) override;
};

bool CommentMatcher::match(string text) {
	if (text.size() == 1 && text[0] == '/') {
		return true;
	}

	if (regex_match(text, _pattern)) {
		if (text.size() > 4 && text[0] == '/' && text[1] == '*') {
			if (text[text.size() - 3] == '*' && text[text.size() - 2] ==
'/') {
				return false;
			}
		}
		return true;
	}

	return false;
}

class CustomMatcher : public Matcher {
private:
	vector<Matcher*> _matchers;

public:
	CustomMatcher(vector<Matcher*> matchers);

	bool match(string text) override;
};

CustomMatcher::CustomMatcher(vector<Matcher*> matchers) {
	_matchers = matchers;
}

bool CustomMatcher::match(string text) {
	for (auto matcher : _matchers) {
		if (matcher->match(text)) {
			return true;
		}
	}

	return false;
}

return true;
```

LexemSplitter.h

```
class LexemSplitter {
public:
	virtual vector<string> splitToLexemes(string text) = 0;
};

class BaseLexemSplitter : public LexemSplitter {
private:
	Matcher* _writtable;
	Matcher* _skippable;

public:
	BaseLexemSplitter(Matcher* writtable, Matcher* skippable);

	vector<string> splitToLexemes(string text) override;
};

BaseLexemSplitter::BaseLexemSplitter(Matcher* writtable, Matcher* skippable) {
	_writtable = writtable;
	_skippable = skippable;
}

vector<string> BaseLexemSplitter::splitToLexemes(string text) {
	vector<string> lexemes;
	string lexem = string();

	for (int i = 0; i < text.size();) {
		if (_skippable->match(lexem + text[i])) {
			for (i; i < text.size(); i++) {
				if (_skippable->match(lexem + text[i]) == false) {
					break;
				}
				lexem += text[i];
			}
			lexem.clear();
		}
		else if (_writtable->match(lexem + text[i])) {
			for (i; i < text.size(); i++) {
				if (_writtable->match(lexem + text[i]) == false) {
					break;
				}
				lexem += text[i];
			}
			lexemes.push_back(lexem);
			lexem.clear();
		}
		else {
			i++;
		}
	}

	return lexemes;
}
```

Token parsers

Definers.h

```
class TypeDefiner {
public:
	virtual bool isType(string word) = 0;
};

class RegexTypeDefiner : public TypeDefiner {
private:
	regex _pattern;

public:
	RegexTypeDefiner(regex pattern);

	bool isType(string word) override;
};

RegexTypeDefiner::RegexTypeDefiner(regex pattern) {
	_pattern = pattern;
}

bool RegexTypeDefiner::isType(string word) {
	return regex_match(word, _pattern);
}

class IdentiferDefiner : public RegexTypeDefiner {
public:
	IdentiferDefiner() : RegexTypeDefiner(regex(R"(^(_|[a-z]|[A-Z])\w*)")) {};
};

class IntegerConstantDefiner : public RegexTypeDefiner {
public:
	IntegerConstantDefiner() : RegexTypeDefiner(regex(R"(\d+)")) {};
};

class FloatConstantDefiner : public TypeDefiner {
private:
	regex _dotFirst = regex(R"(\.\d+)");
	regex _dotBetween = regex(R"(\d+\.\d+)");
	regex _dotLast = regex(R"(\d+\.)");

public:
	bool isType(string word) override {
		return regex_match(word, _dotFirst) || regex_match(word,
_dotBetween) || regex_match(word, _dotLast);
	};
};

class ArrayConstantDefiner : public RegexTypeDefiner {
public:
	ArrayConstantDefiner() : RegexTypeDefiner(regex(R"(\{(\d+, ?)*\d\})")) {};
};
```

TokenParsers.h

```
class TokenParser : public Parser<Token> {
private:
	vector<pair<TypeDefiner*, Type>> _definers;

public:
	TokenParser(vector<pair<TypeDefiner*, Type>> definers);

	Token parse(string word) override;
};

TokenParser::TokenParser(vector<pair<TypeDefiner*, Type>> definers) {
	_definers = definers;
}

Token TokenParser::parse(string word) {
	for (auto pair : _definers) {
		auto definer = pair.first;
		auto type = pair.second;

		if (definer->isType(word)) {
			return Token(type, word);
		}
	}

	return Token(Type::ERROR, word);
}

class TokenParserWithTable : public Parser<Token> {
private:
	ReadOnlyTable<string, Type>* _table;
	Parser<Token>* _baseParser;

public:
	TokenParserWithTable(Parser<Token>* _baseParser, ReadOnlyTable<string,
Type>* table);

	Token parse(string word) override;
};

TokenParserWithTable::TokenParserWithTable(Parser<Token>* baseParser,
ReadOnlyTable<string, Type>* table) {
	_baseParser = baseParser;
	_table = table;
}

Token TokenParserWithTable::parse(string word) {
	if (_table->contains(word)) {
		auto type = _table->get(word);

		return Token(type, word);
	}

	return _baseParser->parse(word);
}
```

Tokenizer

Tokenizer.h

```
class BaseTokenizer : public Parser<vector<Token>> {
private:
	Parser<Token>* _parser;
	LexemSplitter* _splitter;

public:
	BaseTokenizer(Parser<Token>* tokenParser, LexemSplitter* lexemSplitter);

	vector<Token> parse(string text) override;
};

BaseTokenizer::BaseTokenizer(Parser<Token>* tokenParser, LexemSplitter*
lexemSplitter) {
	_parser = tokenParser;
	_splitter = lexemSplitter;
}

vector<Token> BaseTokenizer::parse(string text) {
	vector<Token> tokens;

	for (auto lexem : _splitter->splitToLexemes(text)) {
		tokens.push_back(_parser->parse(lexem));
	}

	return tokens;
}
```