Лабораторная работа № 1.3
«Объектно-ориентированный лексический анализатор»

4 марта 2024 г.

Сергей Виленский, ИУ9-62Б

Цель работы

Целью данной работы является приобретение навыка реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализа.

Индивидуальный вариант

- Идентификаторы: последовательности латинских букв и цифр, начинающиеся с буквы.
- Знаки операций: либо последовательности, состоящие из знаков !, #, $, %, &, *, +, ., /, <, =, >, ?, @, , ^, |, - и ~, либо идентификаторы, записанные в обратных кавычках (например, «‘plus‘»).
- Ключевые слова «where», «->», «=>».
- Комментарии: начинаются с «–» и продолжаются до конца строки, либо ограничены «{-» и «-}», могут занимать несколько строк текста.
- Целочисленные литералы: последовательности десятичных цифр.

Реализация

lib/Compiler/Compiler.cpp

```
#pragma once

#include <iostream>

#include "CompilerAbstract.cpp"
#include "Scanner.cpp"
```

```cpp
namespace Compiler {

class Compiler final : public CompilerAbstract {
public:
    auto GetScanner(std::istream& stream) -> Scanner override {
        return {stream, *this};
    }
};

}  // namespace Compiler
```

lib/Compiler/CompilerAbstract.cpp

```cpp
#pragma once

#include "MessageList.cpp"
#include "NameDictionary.cpp"

namespace Compiler {

class Scanner;

class CompilerAbstract {
public:
    virtual auto GetScanner(std::istream& stream) -> Scanner = 0;

    MessageList Messages;
    NameDictionary Names;
};

}  // namespace Compiler
```

lib/Compiler/Fragment.cpp

```cpp
#pragma once

#include "Position.cpp"

namespace Compiler {

struct Fragment {
    Position Strarting;
    Position Ending;
};

}  // namespace Compiler
```

lib/Compiler/Message.cpp

```cpp
#pragma once

#include <string>

#include "Position.cpp"

namespace Compiler {

struct Message {
    bool IsError;
    std::string Text;
    Position Coord;
};

auto operator<(const Message& leftMsg, const Message& rightMsg) -> bool {
    return leftMsg.Coord.Index < rightMsg.Coord.Index;
}

}  // namespace Compiler
```

lib/Compiler/MessageList.cpp

```cpp
#pragma once

#include <string>
#include <set>

#include "Position.cpp"
#include "Message.cpp"

namespace Compiler {

class MessageList {
public:
    auto AddError(const Position& coord, const std::string& text) -> void {
        _addMessage(true, coord, text);
    }

    auto AddWarning(const Position& coord, const std::string& text) -> void {
        _addMessage(false, coord, text);
    }

    auto GetSorted() -> std::set<Message>& {
        return _messages;
```

```
    }

private:
    auto _addMessage(bool isError, const Position& coord, const std::string& text) -> void {
        _messages.insert({isError, text, coord});
    }

    std::set<Message> _messages{};
};

}  // namespace Compiler
```

lib/Compiler/NameDictionary.cpp

```
#pragma once

#include <string>
#include <vector>

namespace Compiler {

class NameDictionary {
public:
    auto AddName(const std::string& string) -> std::size_t {
        _dict.push_back(string);
        return _dict.size() - 1;
    };

    auto Contains(const std::string& string) const -> bool {
        return std::find(_dict.cbegin(), _dict.cend(), string) != _dict.cend();
    };

    auto GetName(std::size_t code) const -> std::string {
        return _dict.at(code);
    }
private:
    std::vector<std::string> _dict;
};

}  // namespace Compiler
```

lib/Compiler/Position.cpp

```
#pragma once

#include <string>
```

```cpp
#include <iostream>

namespace Compiler {

struct Position {
    std::size_t Line = 1;
    std::size_t Pos = 1;
    std::size_t Index = 0;
};

auto operator<<(std::ostream& output, const Position& position) -> std::ostream& {
    return output
        << '('
        << position.Line
        << ", "
        << position.Pos
        << ')';
}

}  // namespace Compiler
```

lib/Compiler/ProgramaIterator.cpp

```cpp
#pragma once

#include <string>
#include <unordered_map>
#include <cassert>

#include "Position.cpp"

namespace Compiler {

class ProgramaIterator {
public:
    ProgramaIterator(std::string& programa, std::istream& stream)
    : _programa(programa),
      _stream(stream) {
        _programa += _stream.get();
      }

    auto resetProgramaBuffer() -> void {
        _programa.erase(
            std::begin(_programa),
            std::next(std::begin(_programa), _position.Index)
        );
```

```
    _position.Index = 0;
}

auto pos() -> Position& {
    return _position;
}

auto cur() const -> char {
    return _programa.at(_position.Index);
}

auto eof() const -> char {
    return _stream.eof();
}

auto next() -> void {
    assert(!eof());

    if (cur() == '\n') {
        _linesLens.insert({_position.Line, _position.Pos});
        ++_position.Line;
        _position.Pos = 1;
    } else {
        ++_position.Pos;
    }
    ++_position.Index;

    if (_position.Index == _programa.size()) {
        _programa += _stream.get();
    }
}

auto next(std::size_t n) -> void {
    for (std::size_t i = 0; i != n; ++i) {
        next();
    }
}

auto prev() -> void {
    assert(_position.Index != 0);

    --_position.Index;
    if (cur() == '\n') {
        --_position.Line;
        _position.Pos = _linesLens.at(_position.Line);
    } else {
```

```
            --_position.Pos;
        }
    }

    auto prev(std::size_t n) -> void {
        for (std::size_t i = 0; i != n; ++i) {
            prev();
        }
    }

private:
    std::unordered_map<std::size_t, std::size_t> _linesLens;
    std::string& _programa;
    std::istream& _stream;
    Position _position;
};

}  // namespace Compiler
```

lib/Compiler/Scanner.cpp

```
#pragma once

#include <istream>
#include <string>
#include <vector>
#include <memory>

#include "CompilerAbstract.cpp"
#include "TokenIdent.cpp"
#include "TokenOperator.cpp"
#include "TokenKeyword.cpp"
#include "TokenInteger.cpp"
#include "ProgramaIterator.cpp"

namespace Compiler {

class Scanner {
public:
    Scanner(std::istream& stream, CompilerAbstract& compiler)
    : _programa(Program, stream),
      _compiler(compiler) {};

    auto nextToken() -> std::unique_ptr<Token> {
        // пропуск последовательности пробельных символов
        while (!_programa.eof() && isspace(_programa.cur())) {
```

```
    _programa.next();
}

// случай обнаружения однострочного комментария
if (!_programa.eof() && _programa.cur() == '-') {
    _programa.next();
    if (_programa.cur() == '-') {
        _programa.prev();
        Position commentStarts = _programa.pos();

        while (!_programa.eof() && _programa.cur() != '\n') {
            _programa.next();
        }
        if (!_programa.eof()) {
            _programa.next();
        }

        Comments.push_back({commentStarts, _programa.pos()});
        return nextToken();
    } else {
        _programa.prev();
    }
}

// случай обнаружения многострочного комментария
if (!_programa.eof() && _programa.cur() == '{') {
    _programa.next();
    if (_programa.cur() == '-') {
        _programa.prev();
        Position commentStarts = _programa.pos();
        _programa.next();

        while (!_programa.eof()) {
            if (_programa.cur() == '-') {
                _programa.next();
                if (_programa.cur() == '}') {
                    break;
                }
                _programa.prev();
            }
            _programa.next();
        }
        if (!_programa.eof()) {
            _programa.next();
        }
```

```
            Comments.push_back({commentStarts, _programa.pos()});
            return nextToken();
        } else {
            _programa.prev();
        }
    }

    // конец файла
    if (_programa.eof()) {
        return nullptr;
    }

    // ищем максимально длинный префикс программы, описывающий некоторый токен
    Token finishToken{};
    finishToken.Coords.Strarting = finishToken.Coords.Ending = _programa.pos();
    ++finishToken.Coords.Ending.Index;

    std::vector<std::unique_ptr<Token>> tokens{};
    tokens.push_back(std::make_unique<TokenIdent>());
    tokens.back()->Tag = Token::DomainTag::IDENT;
    tokens.push_back(std::make_unique<TokenOperator>());
    tokens.back()->Tag = Token::DomainTag::OPERATOR;
    tokens.push_back(std::make_unique<TokenKeyword>());
    tokens.back()->Tag = Token::DomainTag::KEYWORD;
    tokens.push_back(std::make_unique<TokenInteger>());
    tokens.back()->Tag = Token::DomainTag::INTEGER;
    for (auto& token : tokens) {
        token->Coords.Strarting = token->Coords.Ending = _programa.pos();
    }

    while (!tokens.empty()) {

        // сохранение возможных вариантов
        for (const auto& token : tokens) {
            if (token->isFinished()) {
                finishToken = *token.get();
                finishToken.Coords.Ending = _programa.pos();
            }
        }

        // устранение невалидных далее вариантов
        for (
            auto it = std::crbegin(tokens);
            it != std::crend(tokens);
            ++it
        ) {
```

```
            if (!(*it)->canBePrefix()) {
                tokens.erase((it + 1).base());
            }
        }

        // конец файла
        if (_programa.eof()) {
            break;
        }

        // довавление всем токенам следующего символа программы
        for (auto& token : tokens) {
            token->str += _programa.cur();
        }
        _programa.next();
    }

    // возвращение из лукапа за последний валидный токен
    _programa.prev(_programa.pos().Index - finishToken.Coords.Ending.Index);

    _programa.resetProgramaBuffer();

    // вычисление внутренних аттрибутов
    switch (finishToken.Tag) {
    case Token::DomainTag::NIL: {
        auto returnToken = nextToken();

        // группировка последовательных сообщений об ошибках
        auto errorPos = finishToken.Coords.Strarting;
        auto& messages = _compiler.Messages.GetSorted();
        if (
            messages.size() != 0 &&
            errorPos.Index == std::crbegin(messages)->Coord.Index - 1
        ) {
            messages.erase(--std::cend(messages));
        }
        _compiler.Messages.AddError(
            errorPos,
            "token didnt recognized"
        );

        return returnToken;

    } case Token::DomainTag::IDENT: {
        auto ident = std::make_unique<TokenIdent>();
        ident->Coords = finishToken.Coords;
```

```
        ident->Tag = finishToken.Tag;
        ident->str = finishToken.str;

        // проверка на уникальность идентификатора
        if (_compiler.Names.Contains(ident->str)) {
            _compiler.Messages.AddWarning(
                ident->Coords.Strarting,
                "identificator alredy declared"
            );
        }
        ident->IdentCode = _compiler.Names.AddName(ident->str);
        return ident;

    } case Token::DomainTag::OPERATOR: {
        auto operat = std::make_unique<TokenOperator>();
        operat->Coords = finishToken.Coords;
        operat->Tag = finishToken.Tag;
        operat->str = finishToken.str;
        return operat;

    } case Token::DomainTag::KEYWORD: {
        auto keyword = std::make_unique<TokenKeyword>();
        keyword->Coords = finishToken.Coords;
        keyword->Tag = finishToken.Tag;
        keyword->str = finishToken.str;

        if (keyword->str == "where") {
            keyword->Keyword = TokenKeyword::WHERE;
        } else if (keyword->str == "->") {
            keyword->Keyword = TokenKeyword::ARROW;
        } else if (keyword->str == "=>") {
            keyword->Keyword = TokenKeyword::DOUBLE_ARROW;
        }
        return keyword;

    } case Token::DomainTag::INTEGER: {
        auto integer = std::make_unique<TokenInteger>();
        integer->Coords = finishToken.Coords;
        integer->Tag = finishToken.Tag;
        integer->str = finishToken.str;
        integer->Value = std::stoi(integer->str);
        return integer;

    } default: {
        return nullptr;
    }
```

```
        }
    };

    std::string Program;
    std::vector<Fragment> Comments;

private:
    ProgramaIterator _programa;
    CompilerAbstract& _compiler;
};

}  // namespace Compiler
```

lib/Compiler/Token.cpp

```
#pragma once

#include <string>

#include "Fragment.cpp"

namespace Compiler {

struct Token {
    Fragment Coords;
    enum DomainTag {
        NIL,
        IDENT,
        OPERATOR,
        KEYWORD,
        INTEGER,
    };
    DomainTag Tag = NIL;

    std::string str;

    virtual auto isFinished() const -> bool {return false;};
    virtual auto canBePrefix() const -> bool {return false;};
    virtual ~Token() {};
};

}  // namespace Compiler
```

lib/Compiler/TokenIdent.cpp

```cpp
#pragma once

#include <string>

#include "Token.cpp"

namespace Compiler {

struct TokenIdent final : Token {
    std::size_t IdentCode{};

    auto isFinished() const -> bool override {
        return str.size() > 0 && canBePrefix();
    }

    auto canBePrefix() const -> bool override {
        if (str.size() > 0 && !isalpha(str.at(0))) {
            return false;
        }
        for (char chr : str) {
            if (!isalpha(chr) && !isdigit(chr)) {
                return false;
            }
        }
        return true;
    }
};

}  // namespace Compiler
```

lib/Compiler/TokenInteger.cpp

```cpp
#pragma once

#include <string>

#include "Token.cpp"

namespace Compiler {

struct TokenInteger final : Token {
    std::size_t Value;

    auto isFinished() const -> bool override {
        return canBePrefix() && str.size() > 0;
    }
```

```
    auto canBePrefix() const -> bool override {
        for (char digit : str) {
            if (!isdigit(digit)) {
                return false;
            }
        }
        return true;
    }
};

}  // namespace Compiler
```

lib/Compiler/TokenKeyword.cpp

```
#pragma once

#include <string>

#include "Token.cpp"

namespace Compiler {

struct TokenKeyword final : Token {
    enum DomainKeyword {
        WHERE,
        ARROW,
        DOUBLE_ARROW,
    };
    DomainKeyword Keyword;

    auto isFinished() const -> bool override {
        return str == "where" || str == "->" || str == "=>";
    }

    auto canBePrefix() const -> bool override {
        switch (str.size()) {
        case 0:
            return true;
        case 1:
            return str == "w" || str == "-" || str == "=";
        case 2:
            return str == "wh" || str == "->" || str == "=>";
        case 3:
            return str == "whe";
        case 4:
```

```
            return str == "wher";
        case 5:
            return str == "where";
        default:
            return false;
        }
    }
};

}  // namespace Compiler
```

lib/Compiler/TokenOperator.cpp

```
#pragma once

#include <string>

#include "Token.cpp"

namespace Compiler {

struct TokenOperator final : Token {

    auto isFinished() const -> bool override {
        return canBePrefix() && (
            str.size() == 1 && str.at(0) != '`' ||
            str.size() > 1 && *std::rbegin(str) == '`'
        );
    }

    auto canBePrefix() const -> bool override {
        if (str.size() == 0) {
            return true;
        }
        if (str.size() == 1) {
            char chr = str.at(0);
            return
                chr == '!'  ||
                chr == '#'  ||
                chr == '$'  ||
                chr == '%'  ||
                chr == '&'  ||
                chr == '*'  ||
                chr == '+'  ||
                chr == '.'  ||
                chr == '/'  ||
```

```
                chr == '<'  ||
                chr == '='  ||
                chr == '>'  ||
                chr == '?'  ||
                chr == '@'  ||
                chr == '\\' ||
                chr == '^'  ||
                chr == '|'  ||
                chr == '-'  ||
                chr == '~'  ||
                chr == '`';
        }
        if (str.at(0) != '`') {
            return false;
        }
        if (str.size() >= 2) {
            if (!isalpha(str.at(1))) {
                return false;
            }
        }
        for (std::size_t i = 2; i < str.size() - 1; ++i) {
            if (!isalpha(str.at(i)) && !isdigit(str.at(i))) {
                return false;
            }
        }
        return true;
    }
};

}  // namespace Compiler
```

lab1.3.cpp

```
#include <iostream>
#include <fstream>
#include <memory>

#include "lib/Compiler/Compiler.cpp"

auto main() -> int {
    std::ifstream fileStream{"prog.txt"};

    Compiler::Compiler compiler{};
    auto scanner = compiler.GetScanner(fileStream);

    std::cout << "TOKENS:\n";
```

```
std::unique_ptr<Compiler::Token> nextToken = scanner.nextToken();
while (nextToken != nullptr) {
    std::cout << '\t';
    switch (nextToken->Tag) {
    case Compiler::Token::IDENT:
        std::cout << "IDENT";
        break;

    case Compiler::Token::OPERATOR:
        std::cout << "OPERATOR";
        break;

    case Compiler::Token::KEYWORD:
        std::cout << "KEYWORD";
        break;

    case Compiler::Token::INTEGER:
        std::cout << "INTEGER";
        break;
    }
    std::cout
        << ' '
        << nextToken->Coords.Strarting
        << '-'
        << nextToken->Coords.Ending
        << ": "
        << nextToken->str
        << '\n';

    nextToken = scanner.nextToken();
};

std::cout << "COMMENTS:\n";
for (const auto& comment : scanner.Comments) {
    std::cout
        << '\t'
        << comment.Strarting
        << '-'
        << comment.Ending
        << '\n';
}

std::cout << "MESSAGES:\n";
for (const auto& message : compiler.Messages.GetSorted()) {
    std::cout
        << '\t'
```

```
            << (message.IsError ? "ERROR " : "WRANING ")
            << message.Coord
            << ": "
            << message.Text
            << '\n';
    }

    fileStream.close();
    return 0;
}
```

Тестирование

Входные данные

```
1233 21213
sd sd -- sds123d --
 & + -
 `1232132 2312 312 ds d`
 `token123`
 `13123token`
 ËËË
 3213 {- 123
32131 -}


token
```

Вывод на `stdout`

```
TOKENS:
        INTEGER (1, 1)-(1, 5): 1233
        INTEGER (1, 6)-(1, 11): 21213
        IDENT (2, 1)-(2, 3): sd
        IDENT (2, 4)-(2, 6): sd
        OPERATOR (3, 2)-(3, 3): &
        OPERATOR (3, 4)-(3, 5): +
        OPERATOR (3, 6)-(3, 7): -
        INTEGER (4, 3)-(4, 10): 1232132
        INTEGER (4, 11)-(4, 15): 2312
        INTEGER (4, 16)-(4, 19): 312
        IDENT (4, 20)-(4, 22): ds
        IDENT (4, 23)-(4, 24): d
        OPERATOR (5, 2)-(5, 12): `token123`
        INTEGER (6, 3)-(6, 8): 13123
        IDENT (6, 8)-(6, 13): token
```

```
        INTEGER (8, 2)-(8, 6): 3213
        IDENT (12, 1)-(12, 6): token
COMMENTS:
        (2, 7)-(3, 1)
        (8, 7)-(9, 9)
MESSAGES:
        ERROR (4, 24): token didnt recognized
        WRANING (2, 4): identificator alredy declared
        ERROR (4, 2): token didnt recognized
        WRANING (12, 1): identificator alredy declared
```

Вывод

В результате выполнения данной работы были приобретены навыки реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализа.