Лабораторная работа №2. Ручное построение нисходящих синтаксических анализаторов

Юльцова Наталья М33351

20 сентября 2022 г.

1. Разработка грамматики

Вариант 7: Описание переменных в Си

На вход подаются строки типа : $type_1$ var_1, ... var_k; $type_2$ ...
Бывают простые переменные и указатели(т.е звездочка* переменная). По условию, используем один терминал для переменных и типов - обозначим за v.

Построим грамматику:

$$S \to vFL\ ;S \mid \varepsilon$$
$$L \to\ ,FL \mid \varepsilon$$
$$F \to *F \mid v$$

Нетерминал	Описание
S	Описание переменных
L	Описание переменных через запятую
F	Переменная или указатель

В грамматике нет левой рекурсии или правого ветвления, поэтому она остается без изменений.

2. Построение лексического анализатора

В нашей грамматике есть следующие терминалы:

- * – указатель
- ; – разделяет типы
- , – разделяет переменные
- v – имя переменной или типа
- $ – конец

Построим лексический анализатор. Заведем класс Token для хранения терминалов, в том числе конец строки.

```
public enum Token {
    STAR, SEMICOLON, COMMA, NAME, END
}
```

Терминал	TOKEN
*	STAR
;	SEMICOLON
,	COMMA
v	NAME
$	END

Лексический анализатор:

```
import java.io.IOException;
import java.io.InputStream;
import java.text.ParseException;

public class LexicalAnalyser {
    private final InputStream is;
    private int curChar;
    private int curPos;
    private Token curToken;

    public LexicalAnalyser(InputStream is) throws ParseException {
        this.is = is;
        curPos = 0;
        nextChar();
    }

    private boolean isBlank(int c) {
        return Character.isWhitespace(c);
    }

    private void nextChar() throws ParseException {
        curPos++;
        try {
            curChar = is.read();
        } catch (IOException e) {
            throw new ParseException(e.getMessage(), curPos);
        }
    }

    public String nextToken() throws ParseException {
        while (isBlank(curChar)) {
            nextChar();
```

```
        }

        switch (curChar) {
            case '*':
                nextChar();
                curToken = Token.STAR;
                return "*";
            case ';':
                nextChar();
                curToken = Token.SEMICOLON;
                return ";";
            case ',':
                nextChar();
                curToken = Token.COMMA;
                return ",";
            case '$':
                curToken = Token.END;
                return "$";
            default:
                if (!Character.isAlphabetic(curChar)) {
                    throw new ParseException("Invalid token at pos " + curPos, curPos());
                }
                StringBuilder token = new StringBuilder();
                while (Character.isAlphabetic(curChar) || Character.isDigit(curChar) || curChar
                    token.append(Character.toChars(curChar));
                    nextChar();
                }
                curToken = Token.NAME;
                return token.toString();
        }
    }

    public Token curToken() {
        return curToken;
    }

    public int curPos() {
        return curPos;
    }
}
```

3. Построение синтаксического анализатора

Построим множества FIRST и FOLLOW для нетерминалов нашей грамматики.

Нетерминал	FIRST	FOLLOW
S	v ε	$
L	, ε	;
F	v *	, ;

Заведем структуру для хранения дерева:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Tree {
    String node;
    List<Tree> children;

    public Tree(String node, Tree... children) {
        this.node = node;
        this.children = Arrays.asList(children);
    }

    public Tree(String node) {
        this.node = node;
        children = new ArrayList<>();
    }

    @Override
    public String toString() {
        StringBuilder ans = new StringBuilder();
        for (Tree t : children) {
            ans.append(node).append("->").append(t.node).append(";").append("\n").append(t.toString
        }
        return ans.toString();
    }
}
```

Синтаксический анализатор с использованием рекурсивного спуска:

```
    import java.io.InputStream;
    import java.text.ParseException;

    public class Parser {
        private LexicalAnalyser lex;
        private String token;

        private Tree S() throws ParseException {
            if (lex.curToken() == Token.NAME) {
                String name = token;
                token = lex.nextToken();
                Tree f = F();
                token = lex.nextToken();
                Tree l = L();
                if (lex.curToken() != Token.SEMICOLON) {
                    throw new ParseException("`;` expected at pos " + lex.curPos(), lex.curPos());
                }
                token = lex.nextToken();
                Tree s = S();
                if (lex.curToken() != Token.END) {
```

```
                throw new ParseException("Eof expected at pos " + lex.curPos(), lex.curPos());
            }
            return new Tree("S", new Tree(name), f, l, new Tree("SEMICOLON"), s);
        } else if (lex.curToken() == Token.END) {
            return new Tree("S", new Tree("eps"));
        } else
            throw new ParseException("Expression expected at pos " + lex.curPos(), lex.curPos()
    }

    private Tree L() throws ParseException {
        if (lex.curToken() == Token.SEMICOLON) {
            return new Tree("L", new Tree("eps"));
        } else if (lex.curToken() == Token.COMMA) {
            token = lex.nextToken();
            Tree f = F();
            token = lex.nextToken();
            Tree l = L();
            return new Tree("L", new Tree("COMMA"), f, l);
        }
        throw new ParseException("Expected ; or , at pos " + lex.curPos(), lex.curPos());
    }

    private Tree F() throws ParseException {
        if (lex.curToken() == Token.STAR) {
            token = lex.nextToken();
            Tree f = F();
            return new Tree("F", new Tree("STAR"), f);
        } else if (lex.curToken() == Token.NAME) {
            String name = token;
            return new Tree("F", new Tree(name));
        }
        throw new ParseException("Varname is incorrect. Unexpected token at pos " + lex.curPos(

    }


    public Tree parse(InputStream is) throws ParseException {
        lex = new LexicalAnalyser(is);
        token = lex.nextToken();
        return S();
    }
}
```

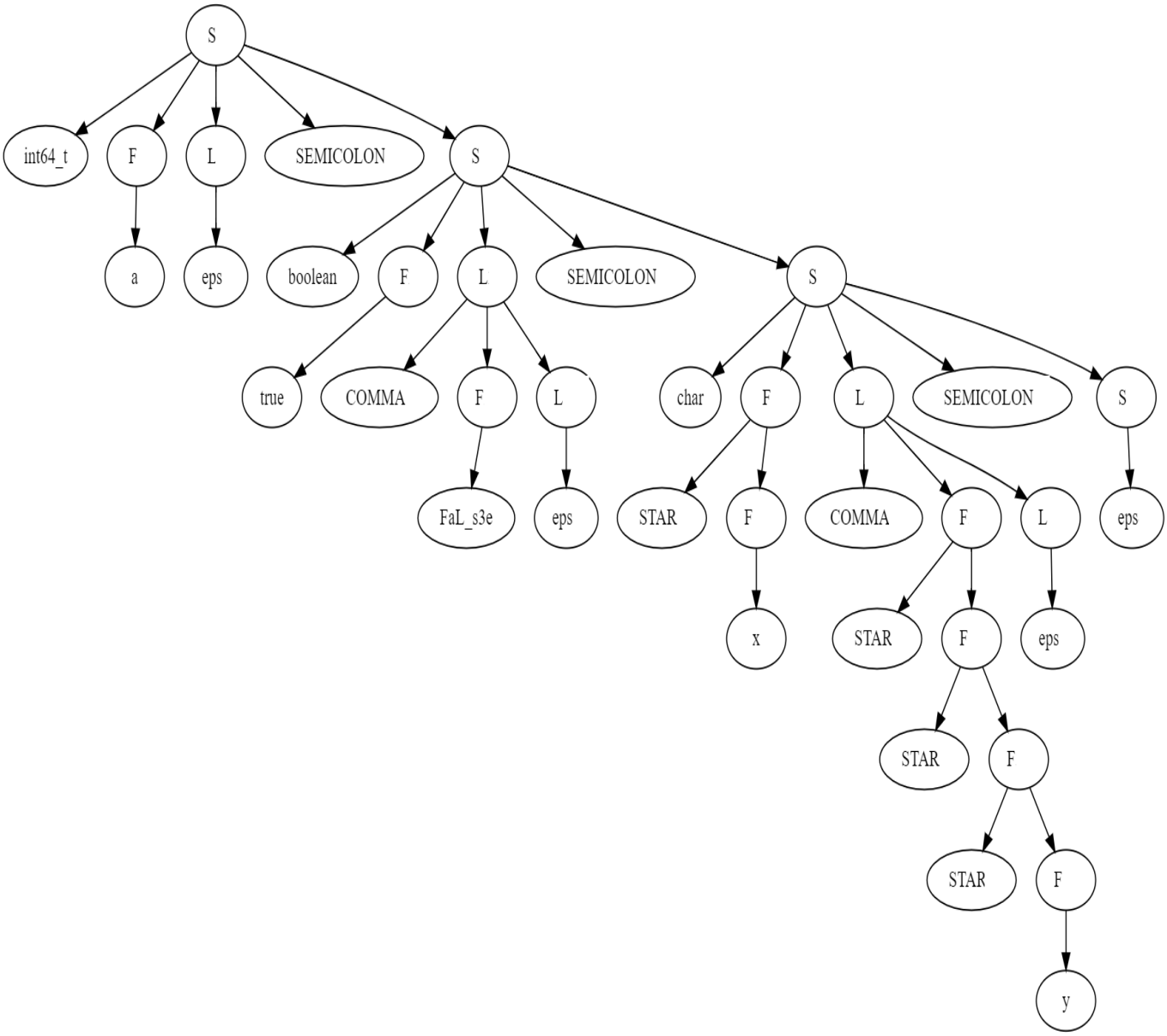
Запуск:

```
import java.text.ParseException
public class Main {
    public static void main(String[] args) throws ParseException {
        final Parser parser = new Parser();
        System.out.println(parser.parse(System.in).toString());
    }
}
```

4. Визуализация дерева разбора

Для изучения результата будем использовать систему GraphViz (https://graphviz.org/) и рассмотримм полученное в ходе тестирования дерево.

Пример на различные правила:
int64_t a;
boolean true, FaL_s3e;
char *x, ***y;$



5. Тестирование

Тест	Описание
	Пустой тест, вернет ошибку
int a;	Простой тест, проверяет $S \to \varepsilon$, $Z \to \varepsilon$ и $N \to v$
int a, b;	Тест на правило $Z \to N, Z$
int64_t a1, b2c;	Тест на распознавание названий переменных и типов
int *a;	Тест на правило $N \to *N$
int **a;	Тест на правило $T \to$: n
int a; boolean b;	Тест на правило $S \to nZN; S$
int a*;	Ошибка - неожиданный символ
64int 1var;	Ошибка
int a	Ожидается ;
a;	Ожидаются имена переменных
int a int b;	Ошибка