Алгоритмизация и программирование

8.5. Разбор выражений

Глухих Михаил Игоревич
mailto: glukhikh@mail.ru

Постановка задачи

- В файле имеется описание функции от **x**:
  - четыре арифметических действия
  - целые константы
  - символ переменной **x**
  - знаки скобок
  - (x + 2) * 4 – 7
  - (5 / (x–3) + 2 * x) * (x – 5)
- Имеется также список значений **x**
- Задача – для каждого значения **x** рассчитать значение функции

```
fun parseExpr(inputName: String,
              values: List<Int>): Map<Int, Int> = TODO()
```

В ходе решения задачи будут рассмотрены

- Теоретические элементы:
  - принципы решения задач лексического и синтаксического анализа
  - принципы разбора и вычисления сложных выражений
- Элементы языка Kotlin:
  - расширения (**extension functions**)
  - перечисления (**enum class**)
  - алгебраические классы (**sealed class**)
  - объекты (**object**)

Разбор текста

- Этапы
  - Лексический анализ – разбиение на атомарные элементы, или **лексемы**

Разбор текста

- Этапы
  - Лексический анализ – разбиение на атомарные элементы, или **лексемы**
  - Синтаксический анализ – разбиение на «предложения», построение деревьев

Разбор текста

- Этапы
  - Лексический анализ – разбиение на атомарные элементы, или **лексемы**
  - Синтаксический анализ – разбиение на «предложения», построение деревьев
  - Семантический анализ – определение «смысла» предложений

Схема анализа текста

Лексемы

- В данной задаче
  - Константы
  - Знак переменной X
  - Знаки операций и скобок

Лексемы

- В данной задаче
  - Константы
  - Знак переменной X
  - Знаки операций и скобок
- Как разбить? Например, через RegEx

Лексический анализ

```
fun String.splitIntoGroups(): List<String> {
    val matchResults = Regex(
        """x|\+|-|\*|/|\(|\)|\d+?| +?|.+?""").
        findAll(this)
    val groups = matchResults.map { it.value }
                             .filter { it.isNotBlank() }
                             .toList()
    return groups
}
```

Лексический анализ

```
fun String.parseExpr(): List<String> {
    val matchResults = Regex(
        """x|\+|-|\*|/|\(|\)|\d+?| +?|.+?""").
        findAll(this)
    val groups = matchResults.map { it.value }
                             .filter { it.isNotBlank() }
                             .toList()
    return groups
}

// (x + 2) * 4 - 7  →
// (, x, +, 2, ), *, 4, -, 7
```

Синтаксический анализ

- Синтаксический анализ, или **парсинг (parsing)** – процесс анализа входной последовательности символов

Синтаксический анализ

- Синтаксический анализ, или **парсинг (parsing)** – процесс анализа входной последовательности символов
  - обычно осуществляется в соответствии с заданной **формальной грамматикой**

Синтаксический анализ

- Синтаксический анализ, или **парсинг (parsing)** – процесс анализа входной последовательности символов
  - обычно осуществляется в соответствии с заданной **формальной грамматикой**

- Синтаксический анализатор, или **парсер (parser)** –часть программы, выполняющая синтаксический анализ

Формальная грамматика

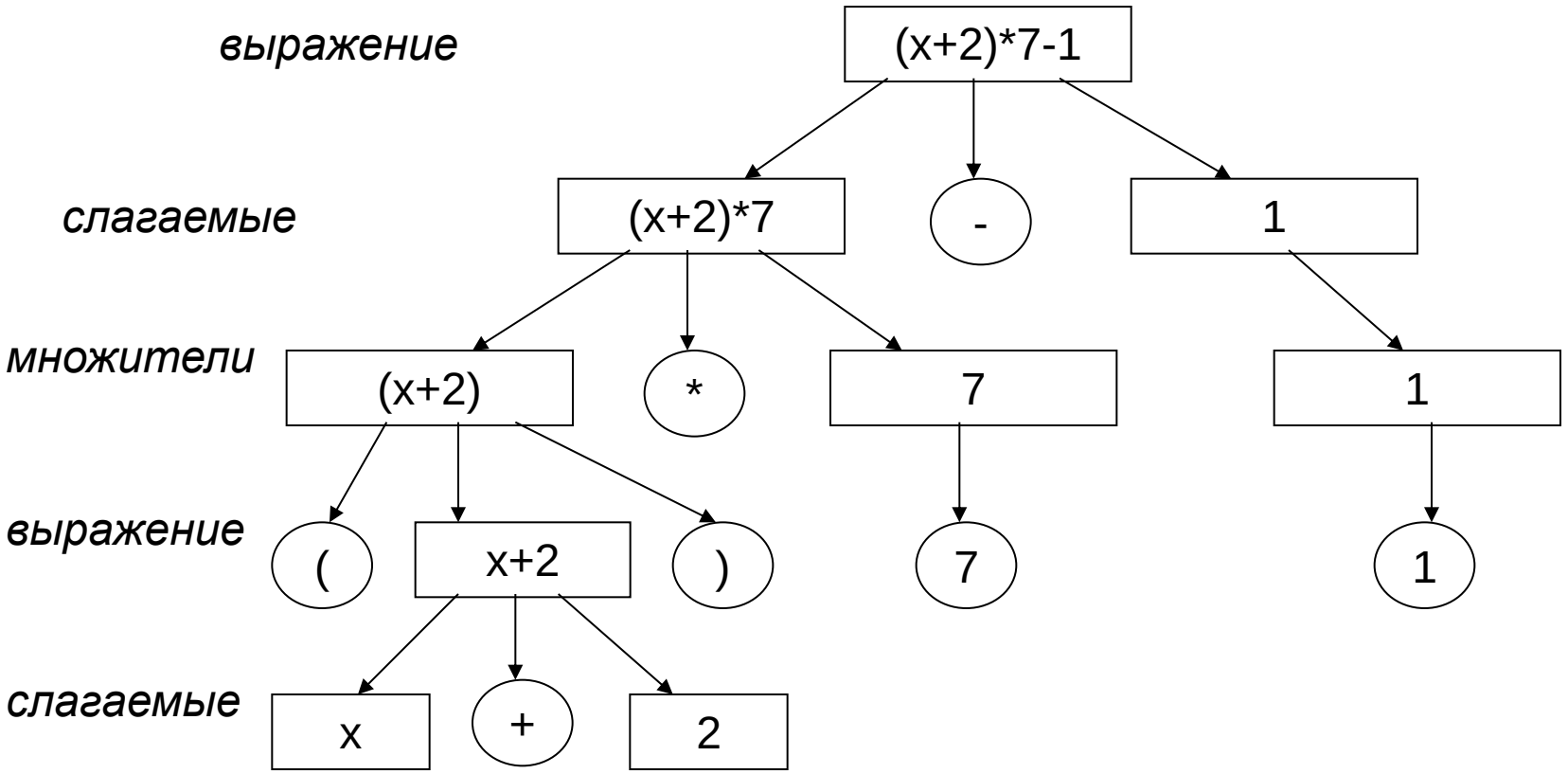
- Формальная грамматика – способ описания формального языка
(в данном случае – языка выражений)

Формальная грамматика

- Формальная грамматика – способ описания формального языка
(в данном случае – языка выражений)

- Например:
  - дано выражение (x + 2) * 7 – 1
  - x – это переменная
  - 2, 7, 1 – это константы
  - (x+2), 7 – это множители
  - x, 2, (x + 2) * 7, 1 – это слагаемые
  - x + 2, (x + 2) * 7 – 1 – это выражения

Грамматическое дерево



Грамматика выражений

<выражение> ::= <слагаемое>

<выражение> ::= <слагаемое> [+-] ... [+-] <слагаемое>

Грамматика выражений

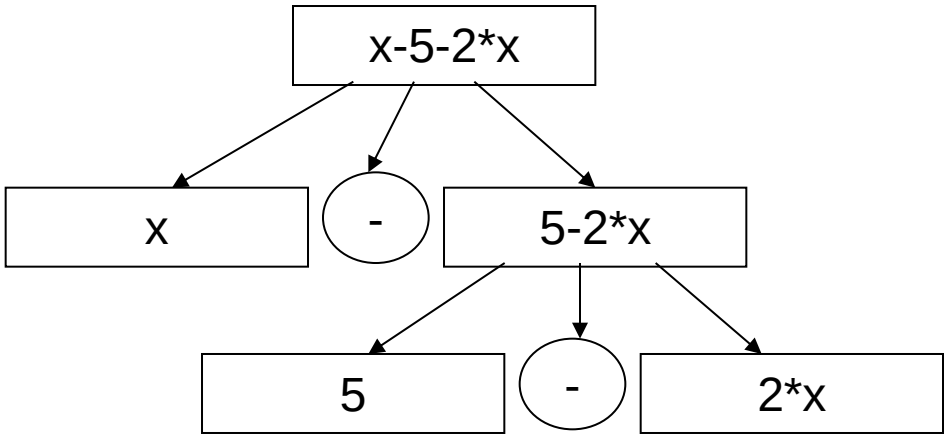
```
<выражение> ::= <слагаемое>

<выражение> ::= <слагаемое> [+-] ... [+-] <слагаемое>

<слагаемое> ::= <множитель>

<слагаемое> ::= <множитель> [*/] ... [*/] <множитель>
```

Грамматика выражений

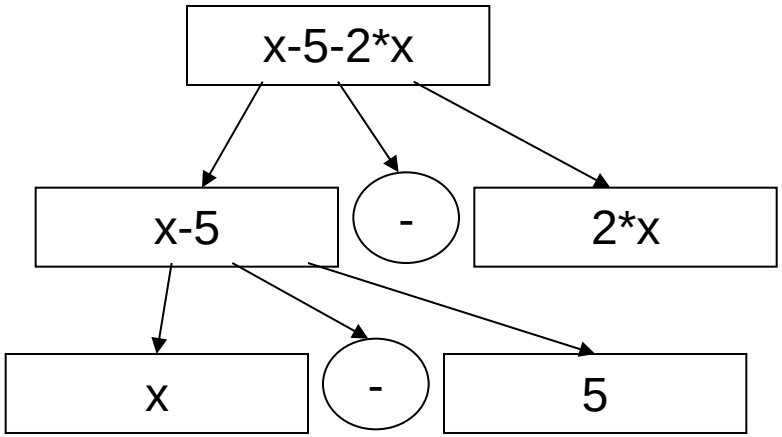
```
<выражение> ::= <слагаемое>

<выражение> ::= <слагаемое> [+-] ... [+-] <слагаемое>

<слагаемое> ::= <множитель>

<слагаемое> ::= <множитель> [*/] ... [*/] <множитель>

<множитель> ::= <переменная>

<множитель> ::= <число>

<множитель> ::= ( <выражение> )
```

Ассоциативность:

правая



левая



Деревья выражений

```
sealed class Expression {
    object Variable : Expression()
    class Constant(val value: Int) : Expression()

    enum class Operation {
        PLUS,
        MINUS,
        TIMES,
        DIV;
    }
    class Binary(
            val left: Expression,
            val op: Operation,
            val right: Expression
    ) : Expression()

    class Negate(val arg: Expression) : Expression()
}
```

Перечисления (enum)

- Тип, перечисляющий все свои возможные значения

```
enum class Operation {
    PLUS,
    MINUS,
    TIMES,
    DIV;
}
```

Алгебраический (sealed) класс

- Тип, перечисляющий все свои разновидности

```
sealed class Expression {
    object Variable : Expression()
    class Constant(val value: Int) : Expression()

    // Operation...

    class Binary(
            val left: Expression,
            val op: Operation,
            val right: Expression
    ) : Expression()

    class Negate(val arg: Expression) : Expression()
}
```

Объект (object)

- Тип с ровно одним возможным значением

```
sealed class Expression {
    object Variable : Expression()
    // …
}
```

Обход дерева + расчёт значения

```
fun Expression.calculate(x: Int): Int = when (this) {
    Variable -> x
    is Constant -> value
    is Binary -> {
        val left  = left.calculate(x)
        val right = right.calculate(x)
        when (op) {
            PLUS  -> left + right
            MINUS -> left – right
            TIMES -> left * right
            DIV   -> left / right
        }
    }
    is Negate -> -arg.calculate(x)
}
```

Разбор + составление дерева

```
class Parser(val groups: List<String>) {
    var pos = 0
    fun parse(): Expression {
        val result = parseExpression()
        if (pos < groups.size) throw IllegalStateException()
        return result
    }
    private fun parseExpression(): Expression {
        var left = parseItem()
        while (pos < groups.size) {
            val op = operationMap[groups[pos]]
            if (op == PLUS || op == MINUS) {
                pos++
                left = Expression.Binary(left, op, parseItem())
            } else return left
        }
        return left
    }
    val operationMap =
            mapOf("+" to PLUS, "-" to MINUS, "*" to TIMES, "/" to DIV)
}
```

Разбор + составление дерева

```
class Parser(val groups: List<String>) {
    var pos = 0
    // ...
    private fun parseItem(): Expression {
        var left = parseFactor()
        while (pos < groups.size) {
            val op = operationMap[groups[pos]]
            if (op == TIMES || op == DIV) {
                pos++
                left = Expression.Binary(
                                left, op, parseFactor())
            } else return left
        }
        return left
    }
}
```

Разбор и составление дерева

```
class Parser(val groups: List<String>) {
    var pos = 0

    // ...
    private fun parseFactor(): Expression =
            if (pos >= groups.size) throw IllegalStateException()
            else {
                val group = groups[pos++]
                when (group) {
                    "x" -> Expression.Variable
                    "-" -> Expression.Negate(parseFactor())
                    "(" -> {
                        val arg = parseExpression()
                        val next = groups[pos++]
                        if (next == ")") arg
                        else throw IllegalStateException()
                    }
                    else -> Expression.Constant(group.toInt())
                }
            }
}
```

Собираем всё вместе

```
fun parseExpr(inputName: String,
              values: List<Int>): Map<Int, Int> {
    val list = File(inputName).readLines().
               firstOrNull()?.splitIntoGroups()
               ?: throw IllegalArgumentException()
    val expr = Parser(list).parse()
    val result = mutableMapOf<Int, Int>()
    for (value in values) {
        result[value] = expr.calculate(value)
    }
    return result
}
```

Всё!