Лабораторная работа №2. Ручное построение нисходящих синтаксических анализаторов

Михайлов Максим, группа M3337

Вариант 9: описание заголовка функции в Kotlin

3 января 2022 г.

1 Построение грамматики

Построим интуитивную грамматику.

$H \to \texttt{fun}\ N \texttt{(} P \texttt{)} R$
$P \to N\ T \texttt{,} P$
$P \to N\ T$
$P \to \varepsilon$
$T \to \texttt{:} N$
$R \to T$
$R \to \varepsilon$

Нетерминал	Описание
H	Заголовок функции
P	Список параметров функции
T	Аннотация типа
R	Возвращаемый тип

В этой грамматике есть правое ветвление для P. Устраним правое ветвление:

$H \to \texttt{fun}\ N \texttt{(} P \texttt{)} R$
$P \to N\ T\ P'$
$P \to \varepsilon$
$P' \to \varepsilon$
$P' \to \texttt{,} P$
$T \to \texttt{:} N$
$R \to T$
$R \to \varepsilon$

Нетерминал	Описание
H	Заголовок функции
P	Список параметров функции
P'	Хвост списка параметров функции
T	Аннотация типа
R	Возвращаемый тип

2 Построение лексического анализатора

Создадим класс `Token` для хранения терминалов.

```
enum class Token(private val humanReadableName: String) {
    FUN("keyword fun"),
    IDENTIFIER("identifier"),
    COLON("colon"),
    LPAREN("left parenthesis"),
    RPAREN("right parenthesis"),
    LANGLE("left angled parenthesis"),
    RANGLE("right angled parenthesis"),
    COMMA("comma"),
    END("end of input");

    override fun toString() = humanReadableName
}
```

Терминал	Токен
`fun`	`FUN`
N	`IDENTIFIER`
`(`	`LPAREN`
`)`	`RPAREN`
`,`	`COMMA`
$	`END`

```
import java.io.IOException
import java.io.InputStream
import java.text.ParseException
import kotlin.properties.Delegates

class LexicalAnalyzer(private var input: InputStream) {
    private var char by Delegates.notNull<Int>()
    var position = -1
        private set
    var identifier = ""
        private set
    lateinit var token: Token
        private set

    init {
```

```
        nextChar()
    }

    private fun nextChar() {
        position++
        try {
            char = input.read()
        } catch (e: IOException) {
            err(e.message ?: "Empty IO exception")
        }
    }

    fun nextToken() {
        while (char.toChar().isWhitespace()) {
            nextChar()
        }
        identifier = ""
        when (char) {
            '('.code -> {
                nextChar()
                token = Token.LPAREN
            }
            ')'.code -> {
                nextChar()
                token = Token.RPAREN
            }
            ','.code -> {
                nextChar()
                token = Token.COMMA
            }
            ':'.code -> {
                nextChar()
                token = Token.COLON
            }
            -1 -> {
                token = Token.END
            }
            '<'.code -> {
                nextChar()
                token = Token.LANGLE
            }
            '>'.code -> {
```

```
                nextChar()
                token = Token.RANGLE
            }
            '`'.code → parseEscapedIdentifier()
            else → parseIdentifierOrFun()
        }
    }

    private fun parseEscapedIdentifier() {
        val sb = StringBuilder()
        addToStringBuilder(sb)
        while (char ≠ '`'.code) {
            if (char == '\r'.code || char == '\n'.code) {
                err("Newline while reading a backtick-escaped identifier")
            }
            if (char == -1) {
                err("End of input while reading a backtick-escaped identifier")
            }
            addToStringBuilder(sb)
        }
        addToStringBuilder(sb)
        setIdentifier(sb.toString())
    }

    private fun parseIdentifierOrFun() {
        val sb = StringBuilder()
        if (!char.toChar().isLetter() && char ≠ '_'.code) {
            err("Expected letter or underscore, found \"${char.toChar()}\"")
        }
        addToStringBuilder(sb)
        while (char.toChar().isLetter() || char == '_'.code || char.toChar().isDigit()) {
            addToStringBuilder(sb)
        }
        sb.toString().also {
            if (it == "fun") {
                token = Token.FUN
            } else {
                setIdentifier(it)
            }
        }
    }
```

```
    private fun addToStringBuilder(sb: StringBuilder) {
        sb.append(char.toChar())
        nextChar()
    }

    private fun setIdentifier(it: String) {
        identifier = it
        token = Token.IDENTIFIER
    }

    private fun err(message: String) {
        throw ParseException(message, position)
    }
}
```

3 Построение синтаксического анализатора

Построим множества FIRST и FOLLOW для нетерминалов нашей грамматики.

Нетерминал	FIRST	FOLLOW
H	`fun`	$
P	ε, N	)
P'	ε, ,	)
T	:	), , , $
R	ε, :	$

Заведем структуру данных для хранения дерева и парсер.

```
import Token.*
import java.io.InputStream
import java.text.ParseException

class Node(val name: String, vararg val children: Node) {
    override fun equals(other: Any?): Boolean {
        if (other !is Node) {
            return false
        }
        return name == other.name && children.contentEquals(other.children)
    }
}
```

```
class Parser {
    private lateinit var lex: LexicalAnalyzer

    private fun h(): Node {
        assertTokenAndAdvance(FUN)
        val functionName = getIdentifier()
        assertTokenAndAdvance(LPAREN)
        val parameters = p()
        assertTokenAndAdvance(RPAREN)
        val returnType = r()
        assertTokens(END)
        return Node("H", Node("fun"), functionName, Node("("), parameters, Node(")"), returnT
    }

    private fun r(): Node {
        return when (lex.token) {
            END -> Node("R")
            COLON -> {
                val typeName = t()
                return Node("R", typeName)
            }
            else -> errTokens(END, COLON)
        }
    }

    private fun pprime(): Node {
        return when (lex.token) {
            RPAREN -> {
                Node("P'")
            }
            COMMA -> {
                lex.nextToken()
                Node("P'", Node(","), p())
            }
            else -> errTokens(RPAREN, COMMA)
        }
    }

    private fun p(): Node {
        return when (lex.token) {
            RPAREN -> {
                Node("P")
```

```
            }
            IDENTIFIER -> {
                val name = getIdentifier()
                val type = t()
                val tail = pprime()
                Node("P", name, type, tail)
            }
            else -> errTokens(RPAREN, IDENTIFIER)
        }
    }

    private fun t(): Node {
        assertTokenAndAdvance(COLON)
        val typeName = getIdentifier()
        val generic = g()
        return Node("T", Node(":"), typeName, generic)
    }

    private fun g(): Node {
        return when (lex.token) {
            LANGLE -> {
                lex.nextToken()
                val typeName = getIdentifier()
                val rest = a()
                lex.nextToken()
                Node("G", Node("<"), typeName, rest, Node(">"))
            }
            RPAREN, COMMA, END -> {
                Node("G")
            }
            else -> errTokens(LANGLE, RPAREN, COMMA, END)
        }
    }

    private fun a(): Node {
        return when (lex.token) {
            COMMA -> {
                lex.nextToken()
                Node("A", aprime())
            }
            RANGLE -> {
                Node("A")
```

```
            }
            else -> errTokens(COMMA, RANGLE)
        }
    }

    private fun aprime(): Node {
        return when (lex.token) {
            COLON -> {
                return Node("A'", getIdentifier(), a())
            }
            RANGLE -> {
                Node("A'")
            }
            else -> errTokens(IDENTIFIER, RANGLE)
        }
    }

    private fun getIdentifier(): Node {
        assertTokens(IDENTIFIER)
        return Node(lex.identifier).also { lex.nextToken() }
    }

    private fun assertTokenAndAdvance(token: Token) {
        assertTokens(token)
        lex.nextToken()
    }

    private fun assertTokens(vararg tokens: Token) {
        if (lex.token !in tokens) {
            errTokens(*tokens)
        }
    }

    fun parse(input: InputStream): Node {
        lex = LexicalAnalyzer(input)
        lex.nextToken()
        return h()
    }

    private fun errTokens(vararg tokens: Token): Nothing {
        err("Expected ${tokens.joinToString(" or ")}, found ${lex.token}")
    }
```

```
    private fun err(message: String): Nothing {
        throw ParseException(message, lex.position)
    }
}
```

4 Визуализация дерева разбора

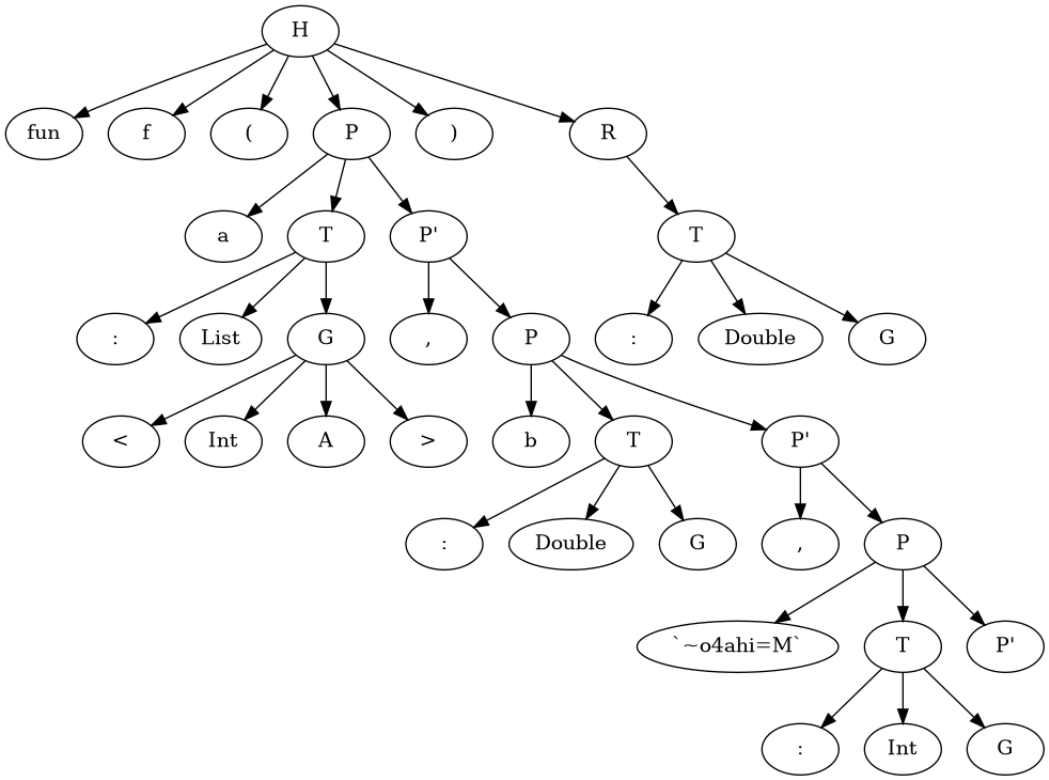

Рис. 1: Пример дерева разбора для
`fun f(a: Int, b: Double, `~o4ahi=M`: Int):Double`

```
import java.io.PrintWriter
import java.nio.file.Path
import kotlin.system.exitProcess

var nodeCount = -1

fun main() {
    // val input = readLine()
    // if (input == null) {
```

```
    //     println("No input, exiting.")
    //     exitProcess(0)
    // }
    val dotFilePath = Path.of("tree.dot")
    dotFilePath.toFile().printWriter().use { out →
        visualize(Parser().parse("fun f(a: List<Int>, b: Double, `~o4ahi=M`: Int): Double".by
    }
    Runtime.getRuntime().exec("dot -Tpng $dotFilePath -o tree.png")
}

private fun visualize(root: Node, out: PrintWriter) {
    out.println("digraph G {")
    dfs(root, out)
    out.println("}")
}

private fun dfs(node: Node, out: PrintWriter, from: Int? = null) {
    nodeCount++
    out.println("$nodeCount [label = \"${node.name}\"]")
    if (from ≠ null) {
        out.println("$from → $nodeCount")
    }
    val curNodeIndex: Int = nodeCount
    for (child in node.children) {
        dfs(child, out, curNodeIndex)
    }
}
```

5 Подготовка набора тестов

Тест	Описание
	Пустой тест (должен произвести ошибку)
`fun f(a: Int, b: Bool): Double`	Небольшой случайный пример
`fun f(a: Int, b: Bool)`	Тест без возвращаемого типа
`fun f(): Double`	Тест без параметров
`fun f()`	Тест без параметров и возвращаемого типа
`fuuun f()`	Тест с неверным ключевым словом `fun`
`fun _F1_a2(`'g[4]VA?`: Int)`	Тест с экранированным идентификатором
`fun f(a: Int):`	Тест с двоеточием для типа, но без типа
`fun f(a: Int): 1`	Тест с неверным идентификатором
`fun f(a: 1)`	Тест с неверным идентификатором
`fun f(1: Int)`	Тест с неверным идентификатором
`fun 1()`	Тест с неверным идентификатором
`fun f(a: Int,)`	Тест с висящей запятой
`fun f(, a: Int)`	Тест с неверно расположенной запятой