

Лабораторная работа №2. Ручное построение нисходящих синтаксических анализаторов

Михайлов Максим, группа М3337

Вариант 9: описание заголовка функции в Kotlin

9 декабря 2021 г.

1 Построение грамматики

Построим интуитивную грамматику.

$H \rightarrow \text{fun } N(P)R$

$P \rightarrow N T, P$

$P \rightarrow N T$

$P \rightarrow \varepsilon$

$T \rightarrow :NG$

$R \rightarrow T$

$R \rightarrow \varepsilon$

$G \rightarrow \langle N A \rangle$

$G \rightarrow \varepsilon$

$A \rightarrow ,N A$

$A \rightarrow ,$

$A \rightarrow \varepsilon$

| Нетерминал | Описание |
|------------|---------------------------|
| H | Заголовок функции |
| P | Список параметров функции |
| T | Аннотация типа |
| R | Возвращаемый тип |
| G | Список генериков |
| A | Элемент списка генериков |

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

$$H \rightarrow \text{fun } N(P)R$$

$$P \rightarrow N T P'$$

$$P \rightarrow \varepsilon$$

$$P' \rightarrow \varepsilon$$

$$P' \rightarrow ,P$$

$$T \rightarrow :NG$$

$$R \rightarrow T$$

$$R \rightarrow \varepsilon$$

$$G \rightarrow \langle N A \rangle$$

$$G \rightarrow \varepsilon$$

$$A \rightarrow ,A'$$

$$A \rightarrow \varepsilon$$

$$A' \rightarrow N A$$

$$A' \rightarrow \varepsilon$$

| Нетерминал | Описание |
|------------|---------------------------|
| H | Заголовок функции |
| P | Список параметров функции |
| T | Аннотация типа |
| R | Возвращаемый тип |
| G | Список женериков |
| A | Элемент списка женериков |

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 |
| <i>N</i> | IDENTIFIER |
| (| LPAREN |
|) | RPAREN |
| < | LANGLE |
| > | RANGLE |
| , | 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 → 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' | $\varepsilon, ,$ |) |
| T | : |), ,, \$ |
| R | $\varepsilon, :$ | \$ |
| G | $\varepsilon, <$ |), ,, \$ |
| A | $\varepsilon, ,$ | > |
| A' | N, 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(")"), returnType)
    }

    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 → {
            val typeName = getIdentifier()
            val rest = a()
            Node("G", typeName, rest)
        }
        RPAREN, COMMA, END → {
            Node("G")
        }
        else → errTokens(LANGLE, RPAREN, COMMA, END)
    }
}

private fun a(): Node {
    return when (lex.token) {
        COMMA → {
            Node("A", aprime())
        }
    }
}

private fun aprime(): Node {
    return when (lex.token) {}
}

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

```
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: Int, b: Double, `~o4ahi=M`: Int): Double".byteInput
    )
    Runtime.getRuntime().exec("dot -Tpng $dotFilePath -o tree.png")
}

private fun visualize(root: Node, out: PrintWriter) {
```

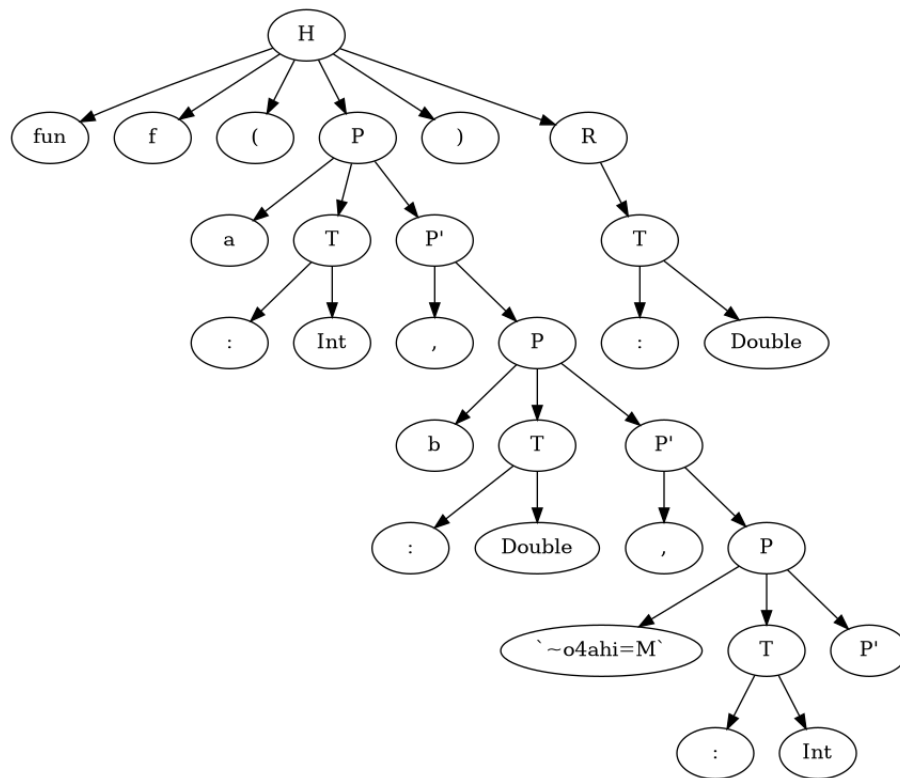


Рис. 1: Пример дерева разбора для
fun f(a: Int, b: Double, ~`o4ahi=M` : Int):Double

```

    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) |
| <code>fun f(a: Int, b: Bool): Double</code> | Небольшой случайный пример |
| <code>fun f(a: Int, b: Bool)</code> | Тест без возвращаемого типа |
| <code>fun f(): Double</code> | Тест без параметров |
| <code>fun f()</code> | Тест без параметров и возвращаемого типа |
| <code>funfun f()</code> | Тест с неверным ключевым словом <code>fun</code> |
| <code>fun _F1_a2(`'g[4]VA?`: Int)</code> | Тест с экранированным идентификатором |
| <code>fun f(a: Int):</code> | Тест с двоеточием для типа, но без типа |
| <code>fun f(a: Int): 1</code> | Тест с неверным идентификатором |
| <code>fun f(a: 1)</code> | Тест с неверным идентификатором |
| <code>fun f(1: Int)</code> | Тест с неверным идентификатором |
| <code>fun 1()</code> | Тест с неверным идентификатором |
| <code>fun f(a: Int,)</code> | Тест с висящей запятой |
| <code>fun f(, a: Int)</code> | Тест с неверно расположенной запятой |