Poročilo

Maj 2025

Kazalo

1 Uvod

Za predmet prevajanje programskih jezikov smo se odločili, da bomo implementirali prevajalnik za naš lastni programski jezik. Ta jezik bo namenjen opisovanju mestne infratrukture in njihovih lastnosti. Naš cilj je ustvariti preprost prevajalnik, ki bo sposoben obdelati osnovne geometrijske konstrukte in jih pretvoriti v GeoJSON format.

2 Osnova

Zahtevano je, da lahko z uporabo vašega jezika opišete geometrijske strukture, točke in ceste v mestu. Torej je potrebno podpreti polilinije (polyline) in poligone (polygon). Na primer, majhen del mesta bi lahko opisali takole:

```
let @p = (1,1.12);
let @q = (2,2);

city ["Maribor"]{
    road["Ptujska cesta"]{
        polyline[(1,2.5),(3.1,4),(5,6)];
    };
    building["FERI"]{
        polygon[(1,1),(1,2),(2,2.7),(2,1),(1,1)];
    };
    area["Igrišče"]{
        polygon[(2,3),(1,1),(5,7),(2,3)];
    };
    lake["Jezero Bled"]{
        circle[(4,3),2.8];
    };
    park["Park Maribor"]{
        circle[(5,9),2];
    };
    road["Ljubljanska cesta"]{
        polyline[$p,$q,$p+$q];
    };
}

?{[(1,1),$p,$q,(3,4)],[(1,1),3]};
let @r = (fst(p),snd(q));
```

2.1 Konstrukti

Uporabljen jezik vsebuje naslednje konstrukte.

2.1.1 Enota

Naša nevtralna enota je: `null`

2.1.2 Realna števila

Za vrednosti imamo realna števila: `1.0`, `2.5`, `-3.14`, `0.0`

2.1.3 Nizi

Za nize uporabljamo dvojne narekovaje, najdemo jih lahko v oglatih oklepajih: `"Ptujska cesta"`, `"FERI"`, `"Park"`

2.1.4 Koordinate

S koordinatami lahko predstavimo lokacije na zemljevidu, v tem primeru je prva komponenta longituda, druga komponenta pa je latituda. `(1.0,2.5)`, `(3.1,4)`, `(5,6)`

2.1.5 Bloki

Blok je sestavljen iz imena objekta, ki ga želimo definirati, definiramo jih lahko s “polygon”, “polyline” in “circle”. Blok se zaključi z okroglimi oklepaji.

2.1.5.1 City

Blok “City” je osrednji blok, ki ga uporabljamo za definiranje mesta. Vsebuje lahko druge bloke, kot so ceste, stavbe, območja, parki in jezera.

```
city["IME"]{
    BLOKI
};
```

2.1.5.2 Road

“Road” je blok, ki ga uporabljamo za definiranje ceste. Vsebuje lahko ukaze za izris, ki izrišejo črte.

```
road["IME"]{
    COMMANDS
};
```

2.1.5.3 Building

“Building” je blok, ki ga uporabljamo za definiranje stavb. Vsebuje lahko ukaze za izris stavbe.

```
building["IME"]{
    COMMANDS
};
```

2.1.5.4 Area

“Area” je blok, ki ga uporabljamo za definiranje območij. Vsebuje lahko ukaze za izris območij.

```
area["IME"]{
    COMMANDS
};
```

2.1.6 Ukazi

Bloki vsebujejo ukaze, ki jih lahko uporabimo za izris geometrijskih oblik. Ukazi so lahko “polyline”, “polygon” in “circle”.

2.1.6.1 Polyline

“Polyline” je ukaz, ki ga uporabljamo za izris polilinij. Vsebuje lahko koordinate za točke, med katerimi so izrisane črte. Končna in začetna točki sta lahko poljubni.

```
polyline[TOČKA,TOČKA,TOČKA];
```

2.1.6.2 Polygon

"Polygon" je ukaz, ki ga uporabljamo za izris poligonov. Vsebuje lahko koordinate za točke, med katerimi so izrisane črte. Poligon se mora zaključi z začetno točko.

```
polygon[TOČKA1,TOČKA2,TOČKA3,TOČKA1];
```

2.1.6.3 Circle

"Circle" je ukaz, ki ga uporabljamo za izris krogov. Vsebuje lahko koordinato za središče in polmer kroga.

```
circle[TOČKA, RADIJ];
```

3 Nadgradja

Jezik smo nadgradili z validacijo, dodatnimi elementi, spremenljivkami, izjavami in povpraševanji.

3.1 Validacija

Validirali bomo poligone pri čemer bomo preverili, da se prvi in zadnji točki ujemata, pri tem morajo biti vse točke v poligonu različne (razen prve in zadnje). Hkrati bomo preverjali, da se črte ob izrisu poligona ne prekivajo ali sekajo. Ob pojavitvi napake bomo o tem obvestili uporabnika.

```
polygon[(1,1),(3,3),(3,1),(1,3),(1,1)];
```

3.2 Dodatni elementi

Na zemljevidu lahko definiramo tudi dodatne elemente, ki jih lahko uporabljamo za definicijo mesta.

3.2.1 Lake

"Lake" je blok, ki ga uporabljamo za definiranje jezer. Vsebuje lahko ukaze za izris jezer.

```
lake["Jezero Bled"]{
    COMMANDS
};
```

3.2.2 Park

"Park" je blok, ki ga uporabljamo za definiranje parkov. Vsebuje lahko ukaze za izris parkov.

```
park["IME"]{
    COMMANDS
};
```

3.3 Spremenljivke

V jeziku lahko definiramo spremenljivke, ki jih lahko uporabljamo za shranjevanje vrednosti. Spremenljivke se definirajo z uporabo ključe "let", in predstavlja točko, ki jo definiramo v oklepajih. Pred samim imenom spremenvljicke mora biti znak `$`.

```
let @IME = (KOORDINATA X,KOORDINATA Y);
```

3.4 Izjave

Podprte izjave v jeziku vključujejo: seštevanje, odštevanje, dostop do prve koordinate in dostop do druge koordinate.

3.4.1 Seštevanje

Seštevanje dveh točk se izvede tako, da se seštejeta obe komponenti točk. Rezultat je nova točka.

```
polyline[(3,4),(2,1)+(3,4)];
```

3.4.2 Odštevanje

Odštevanje dveh točk se izvede tako, da se odštejeta obe komponenti točk. Rezultat je nova točka.

```
polyline[(1,1),$q,(2,4)-$q];
```

3.4.3 First & Second

Dostop do prve in druge komponente točke se izvede tako, da se uporabita funkciji `fst` in `snd`.

```
let @r = (fst($p),snd($q));
```

3.5 Povpraševanja

Povpraševanje je določeno z množico točk, zapisanih v oglatih oklepajih. Poleg te množice je podana dodatna točka z določenim radijem, ki opredeljuje krožno območje povpraševanja. Rezultat povpraševanja je množica vseh točk iz začetne množice, ki ležijo znotraj tega krožnega območja.

```
?{[TOČKA1,TOČKA2,TOČKA3,TOČKA4],[TOČKA0,RADIJ]};
```

4 Gramatika z BNF notacijo

```
izraz   ::= izraz** izraz*
izraz*  ::= izraz izraz* | ε
izraz** ::= SPREMENLJIVKA_DEF | POVPRAŠEVANJE | CITY

// definicija mesta
CITY ::= city["IME"] { BLOCKS }

// definicija bloka
BLOCKS ::= BLOCK BLOCKS*
BLOCKS* ::= BLOCK BLOCKS* | ε
BLOCK ::= ROAD | BUILDING | AREA | LAKE | PARK

ROAD ::= road["IME"] { POLYLINE }
BUILDING ::= building["IME"] { IZRIS }
AREA ::= area["IME"] { IZRIS }
LAKE ::= lake["IME"] { IZRIS }
PARK ::= park["IME"] { IZRIS }

IZRIS ::= POLYGON | KROG

// definicija spremenljivke
```

```
SPREMENLJIVKA_DEF ::= let @ IME = TOČKA;

POLYLINE ::= polyline[TOČKE];
POLYGON ::= polygon[TOČKE];

KROG ::= circle KROG*
KROG* ::= [TOČKA, KROG**]
KROG** ::= FIRST_SECOND | ŠTEVILO

// definicija povpraševanja
POVPRAŠEVANJE ::= ?{[TOČKE],KROG*};

//definicija večih točk
TOČKE ::= TOČKA TOČKE*
TOČKE* ::= , TOČKA TOČKE* | ε

// definicija točke
TOČKA ::= TOČKA** TOČKA*
TOČKA* ::= OPERACIJA TOČKA** TOČKA* | ε
TOČKA** ::= ( KOORDINATA , KOORDINATA )  | SPREMENLJIVKA PERACIJA

KOORDINATA ::= ŠTEVILO | FIRST_SECOND

OPERACIJA ::= + | -

// definicija FIRST in SECOND
FIRST_SECOND ::= fst TOČKA | snd TOČKA

// uporaba spremenljivke
SPREMENLJIVKA ::= $IME

ŠTEVILO ::= [0-9] ŠTEVILO*
ŠTEVILO* ::= [0-9] ŠTEVILO* | . REALNO | ε

REALNO ::= [0-9] REALNO*
REALNO* ::= [0-9] REALNO* | ε

// definicija niza
IME ::= [a-zA-Z_] IME*
IME* ::= [a-zA-Z0-9_] IME* | ε
```

5 Izračun FIRST in FOLLOW množic

5.1 Izračun FIRST množic

```
FIRST(IZRAZ) = { let, ?, city }
FIRST(IZRAZ*) = { let, ?, city, ε }
FIRST(IZRAZ**) = { let, ?, city }
FIRST(CITY) = { city }
FIRST(BLOCKS) = { road, building, area, lake, park }
FIRST(BLOCKS*) = { road, building, area, lake, park, ε }
```

```
FIRST(ROAD) = { road }
FIRST(BUILDING) = { building }
FIRST(AREA) = { area }
FIRST(LAKE) = { lake }
FIRST(PARK) = { park }
FIRST(IZRIS) = { polygon, circle}
FIRST(SPREMENLJIVKA_DEF) = { let}
FIRST(POLYLINE) = { polyline }
FIRST(POLYGON) = { polygon }
FIRST(KROG) = { circle }
FIRST(KROG*) = { [ }
FIRST(KROG**) = { fst, snd, [0-9] }
FIRST(POVPRAŠEVANJE) = { ? }
FIRST(TOČKE) = { ( , $ }
FIRST(TOČKE*) = { ,  ε }
FIRST(TOČKA) = { ( , $ }
FIRST(TOČKA*) = { + , - }
FIRST(TOČKA**) = { ( , $ }
FIRST(KOORDINATA) = { [0-9] , fst , snd}
FIRST(OPERACIJA) = { + , - }
FIRST(FIRST_SECOND) = { fst, snd }
FIRST(SPREMELNJIVKA) = { $ }
FIRST (ŠTEVILO) = { [0-9]}
FIRST (ŠTEVILO*) = { [0-9], . , ε }
FIRST(REALNO) = { [0-9] }
FIRST(REALNO*) =  { [0-9], ε }
FIRST(IME) = { [a-zA-Z0-9_] }
FIRST(IME*) = { [a-zA-Z0-9_], ε }
```

5.2 Izračun FOLLOW množic

```
FOLLOW(IZRAZ) = { let, ?, city, ε }
FOLLOW(IZRAZ*) = { let, ?, city, ε }
FOLLOW(IZRAZ**) = { let, ?, city, ε }
FOLLOW(CITY) = { let, ?, city, ε }
FOLLOW(BLOCKS) = { } }
FOLLOW(BLOCKS*) = { } }
FOLLOW(BLOCK) = { road, building, area, lake, park, }
FOLLOW(ROAD) = { road, building, area, lake, park, }
FOLLOW(BUILDING) = { road, building, area, lake, park, }
FOLLOW(AREA) = { road, building, area, lake, park, }
FOLLOW(LAKE) = { road, building, area, lake, park, }
FOLLOW(PARK) = { road, building, area, lake, park, }
FOLLOW(IZRIS) = { } }
FOLLOW(SPREMENLJIVKA_DEF) = { let, ?, city, ε }
FOLLOW(POLYLINE) = { } }
FOLLOW(POLYGON) = { } }
FOLLOW(KROG) = { } }
FOLLOW(KROG*) = { } }
FOLLOW(KROG**) = { ] }
FOLLOW(POVPRAŠEVANJE) = { let, ?, city, ε }
FOLLOW(TOČKE) = { ] }
```

```
FOLLOW(TOČKE*) = { ] }
FOLLOW(TOČKA) = { ; ,, ε, )}
FOLLOW(TOČKA*) = { ; ,, ε, )}
FOLLOW(TOČKA**) = { + , -}
FOLLOW(KOORDINATA) = { , ) }
FOLLOW(OPERACIJA) = { TOČKA }
FOLLOW(FIRST_SECOND) = { , ) }
FOLLOW(SPREMELNJIVKA) =  { + , - , ε }
FOLLOW(ŠTEVILO) = { , ) }
FOLLOW(ŠTEVILO*) = { , ) }
FOLLOW(REALNO) = { , )}
FOLLOW(REALNO*) = { , ) }
FOLLOW(IME) = { let, ?, city, ε }
FOLLOW(IME*) = { let, ?, city, ε }
```

6 Implementacija abstraktnega sintaktičnega drevesa

```
sealed interface Expr

data class VariableDefinition(val name: String, val value: Point) : Expr
data class City(val name: String, val blocks: List<Block>) : Expr
data class Query(val points: List<Point>, val center: Point, val radius: Expr) : Expr

// --- Izrazi ---
data class NumberLiteral(val value: Double) : Expr
data class VariableReference(val name: String) : Expr
data class FirstCoordinate(val point: Expr) : Expr
data class SecondCoordinate(val point: Expr) : Expr

// --- Bloki v mestu ---
sealed interface Block
data class Road(val name: String, val polyline: Polyline) : Block
data class Building(val name: String, val shape: Shape) : Block
data class Area(val name: String, val shape: Shape) : Block
data class Lake(val name: String, val shape: Shape) : Block
data class Park(val name: String, val shape: Shape) : Block

// --- Oblike ---
sealed interface Shape
data class Polygon(val points: List<Point>) : Shape
data class Circle(val center: Point, val radius: Expr) : Shape

// --- Polylines ---
data class Polyline(val points: List<Point>)

// --- Točke ---
data class Point(val components: List<PointComponent>)

sealed interface PointComponent
data class CoordinatePair(val x: Expr, val y: Expr) : PointComponent
data class VariableRef(val name: String) : PointComponent
```

```
data class First(val point: Point) : PointComponent
data class Second(val point: Point) : PointComponent
```

S temi razredi definiramo vse kar potrebujemo, torej: definiranje spremenljivk, mesta, poizvedb. Nato imamo izraze, kot so: števila, uporaba spremenljivk ter dostop do prve in druge koordinate. Naslednji so elementi mesta, to so: cesta, zgradba, območja, jezera in park. Nato imamo razred za oblike, kot so: poligon, krog, "polyline" (črta) ter točka, ki je definirana iz več komponent. Te komponente so: par koordinat (x in y), referenca na spremenljivko ter prva in druga komponenta točke, ki se uporablja da izvlečeta ali prvo ali drugo koordinato iz točke.

7 Implementacija pregledovalnika (scanner)

```
class Scanner(private val input: String) {
    private var index = 0
    private var line = 1
    private var column = 1
    private val tokens = mutableListOf<Token>()
    private var current = 0

    init {
        scanTokens()
    }
```

Na začetku pregledovalnika kot parameter vzamemo niz, ki ga bo pregledovalnik razčlenjeval. Kot parameter pregledovalnika imamo `input`, ki je niz, ki ga bo Scanner razčlenjeval. Nato imamo naslednje spremenljivke:

- index: trenutna pozicija v nizu.
- line/column: trenutno vrstico in stolpec, za lažje razhroščevanje in iskanje napak.
- tokens: seznam vseh prepoznanih žetonov.
- current: indeks trenutnega položaja v `tokens` seznamu.

Takoj ob kreiranju Scanner razreda se tudi pokliče scanTokens() in napolni seznam `tokens`.

```
private fun scanTokens() {
        while (index < input.length) {
            val ch = input[index]
            when (ch) {
                ' ', '\r', '\t' -> advance()
                '\n' -> advanceLine()

                '{' -> addToken(TokenType.LBRACE, "{")
                '}' -> addToken(TokenType.RBRACE, "}")
                '[' -> addToken(TokenType.LBRACKET, "[")
                ']' -> addToken(TokenType.RBRACKET, "]")
                '(' -> addToken(TokenType.LPAREN, "(")
                ')' -> addToken(TokenType.RPAREN, ")")
                ',' -> addToken(TokenType.COMMA, ",")
                ';' -> addToken(TokenType.SEMICOLON, ";")
                '=' -> addToken(TokenType.EQUAL, "=")
                '+' -> addToken(TokenType.PLUS, "+")
```

```
                '-' -> addToken(TokenType.MINUS, "-")
                '?' -> addToken(TokenType.QUESTION_MARK, "?")
                '@' -> addToken(TokenType.AT, "@")
                '$' -> addToken(TokenType.DOLLAR, "$")
                '"' -> {
                    val startColumn = column
                    advance() // skip opening "
                    val str = readWhile { it != '"' }
                    advance() // skip closing "
                    tokens.add(Token(TokenType.STRING, str, line, startColumn))
                }
                in '0'..'9' -> {
                    val startColumn = column
                    val number = readNumber()
                    tokens.add(Token(TokenType.NUMBER, number, line, startColumn))
                }
                else -> {
                    if (ch.isLetter() || ch == '_') {
                        val startColumn = column
                        val word = readWhile { it.isLetterOrDigit() || it == '_' }
                        val type = when (word) {
                            "city" -> TokenType.CITY
                            "road" -> TokenType.ROAD
                            "building" -> TokenType.BUILDING
                            "area" -> TokenType.AREA
                            "lake" -> TokenType.LAKE
                            "park" -> TokenType.PARK
                            "polyline" -> TokenType.POLYLINE
                            "polygon" -> TokenType.POLYGON
                            "circle" -> TokenType.CIRCLE
                            "let" -> TokenType.LET
                            "fst" -> TokenType.FST
                            "snd" -> TokenType.SND
                            else -> TokenType.ID
                        }
                        tokens.add(Token(type, word, line, startColumn))
                    } else {
                        throw IllegalArgumentException("Unexpected character:  '$ch'
                        at line $line, column $column")
                    }
                }
            }
        }
        tokens.add(Token(TokenType.EOF, "", line, column))
    }
```

V funkciji scanTokens() imamo najprej glavno zanko, ki preglejuje vsak znak, dokler ne pridemo do konca. V spremenljivko ch shranimo trenuten znak, ki ga bomo obdelali. Najprej preskočimo vse nepomembne znake, kot so presledki, tabulatorji, nato pa še preverimo za nove vrstice, in tudi za te pravilno poskrbimo. Nato za vsaki znak dodamo žeton z ustreznim tipom. Ko naletimo na narekovaje, potem za žeton dodamo niz ki je med začetnim in končnim narekovajem. Za števke med 0 in 9

uporabimo funkcijo ki nam bo prebrala število. Na koncu še preverjamo če je beseda ena izmed rezerviranih besed, ki jih uporabljamo, če ni, potem se ta beseda smatra kot ime spremenljivke (ID). Ujemamo tudi napake, če znaka ne prepoznamo. Na koncu funkcije še dodamo znak za konec datoteke.

```
private fun advance() {
    index++
    column++
}

private fun advanceLine() {
    index++
    line++
    column = 1
}
```

Funkcija advance() je namenjena, da se premaknemo naprej za en znak, v isti vrstici. Funkcija advanceLine() pa se premakne naprej v novo vrstico.

```
private fun addToken(type: TokenType, lexeme: String) {
    tokens.add(Token(type, lexeme, line, column))
    advance()
}
```

Funkcija addToken() sprejme vrsto žetona in dejanski niz, ter ga doda v `tokens` seznam.

```
fun nextToken(): Token {
    return if (current < tokens.size) tokens[current++] else tokens.last()
}
```

Funkcija nextToken() vrne naslednji žeton, če pa smo prišli do konca, potem vrne zadnjega, torej EOF (end of file).

```
private fun readWhile(condition: (Char) -> Boolean): String {
    val start = index
    while (index < input.length && condition(input[index])) {
        advance()
    }
    return input.substring(start, index)
}
```

Funkcija readWhile() se uporablja takrat ko se želimo premikati po `input` oz. vnosu dokler ne zadostuje nekemu pogoju (condition). Na koncu tudi vrne podniz ki ustreza pogoju.

```
private fun readNumber(): String {
    val start = index
    while (index < input.length && input[index].isDigit()) {
        advance()
    }
    if (index < input.length && input[index] == '.') {
        advance()
        while (index < input.length && input[index].isDigit()) {
```

```
            advance()
        }
    }
    return input.substring(start, index)
}
```

Funkcija readNumber() bere cela in decimalna števila, in prebrano število tudi vrne.

8 Implementacija razčlenjevalnika (parser)

```
class Parser(private val scanner: Scanner) {

    private var currentToken: Token = scanner.nextToken()
```

Kot parameter razčlenjevalnika damo pregledovalnik, s spremenljivko currentToken pa spremljamo kateri žeton trenutno obdelujemo.

```
private fun readToken(expectedType: TokenType): Token {
        if (currentToken.type == expectedType) {
            val token = currentToken
            currentToken = scanner.nextToken()
            return token
        } else {
            throw IllegalArgumentException("Expected token $expectedType but got
${currentToken.type} at line ${currentToken.line}, at column ${currentToken.column}
with lexeme '${currentToken.lexeme}'")
        }
    }
```

Funkcija readToken() preveri ali je trenutni žeton pravilen, če ni, potem vrne napako, drugače prebere naslednjega.

```
fun parseExpr(): List<Expr> {
        val exprs = mutableListOf<Expr>()
        while (currentToken.type in listOf(LET, QUESTION_MARK, CITY)) {
            val expr = when (currentToken.type) {
                LET -> parseVariableDef()
                QUESTION_MARK -> parseQuery()
                CITY -> parseCity()
                else -> throw IllegalArgumentException("Unexpected token
${currentToken.type}")
            }
            exprs.add(expr)
        }
        return exprs
    }
```

Funkcija parseExpr() je glavna funkcija, ki obdeluje seznam izrazov.

```
private fun parseCity(): City {
        readToken(CITY)
        readToken(LBRACKET)
```

```
        val name = readToken(STRING).lexeme
        readToken(RBRACKET)
        readToken(LBRACE)
        val blocks = parseBlocks()
        readToken(RBRACE)
        return City(name, blocks)
    }
```

Funkcija parseCity() obdeluje mesta, ime mesta ter notranje bloke znotraj mesta.

```
private fun parseBlocks(): List<Block> {
        val blocks = mutableListOf<Block>()
        while (currentToken.type in listOf(ROAD, BUILDING, AREA, LAKE, PARK)) {
            val block: Block = when (currentToken.type) {
                ROAD -> parseRoad()
                BUILDING -> parseBuilding()
                AREA -> parseArea()
                LAKE -> parseLake()
                PARK -> parsePark()
                else -> throw IllegalArgumentException("Unexpected block type:
${currentToken.type}")
            }
            blocks.add(block)
        }
        return blocks
    }
```

Funkcija parseBlocks() obdeluje vsak blok, ki ima svojo funkcijo.

```
private fun parseRoad(): Road {
        readToken(ROAD)
        readToken(LBRACKET)
        val name = readToken(STRING).lexeme
        readToken(RBRACKET)
        readToken(LBRACE)
        val polyline = parsePolyline()
        readToken(RBRACE)
        readToken(SEMICOLON)
        return Road(name, polyline)
    }
```

Funkcija parseRoad() obdeluje ceste, ime ceste ter njeno obliko (polyline).

```
private fun parseBuilding(): Building {
        readToken(BUILDING)
        readToken(LBRACKET)
        val name = readToken(STRING).lexeme
        readToken(RBRACKET)
        readToken(LBRACE)
        val shape = parseShape()
        readToken(RBRACE)
        readToken(SEMICOLON)
```

```
        return Building(name, shape)
    }

    private fun parseArea(): Area {
        readToken(AREA)
        readToken(LBRACKET)
        val name = readToken(STRING).lexeme
        readToken(RBRACKET)
        readToken(LBRACE)
        val shape = parseShape()
        readToken(RBRACE)
        readToken(SEMICOLON)
        return Area(name, shape)
    }

    private fun parseLake(): Lake {
        readToken(LAKE)
        readToken(LBRACKET)
        val name = readToken(STRING).lexeme
        readToken(RBRACKET)
        readToken(LBRACE)
        val shape = parseShape()
        readToken(RBRACE)
        readToken(SEMICOLON)
        return Lake(name, shape)
    }

    private fun parsePark(): Park {
        readToken(PARK)
        readToken(LBRACKET)
        val name = readToken(STRING).lexeme
        readToken(RBRACKET)
        readToken(LBRACE)
        val shape = parseShape()
        readToken(RBRACE)
        readToken(SEMICOLON)
        return Park(name, shape)
    }
```

Po istem postopku to velja tudi za zgradbe, območja, jezera ter parke.

```
private fun parseShape(): Shape {
        return when (currentToken.type) {
            POLYGON -> parsePolygon()
            CIRCLE -> parseCircle()
            else -> throw IllegalArgumentException("Expected POLYGON or CIRCLE but got
${currentToken.type}")
        }
    }
```

Funkcija parseShape() loči med poligonom in krogom. V primeru napaka to tudi vrže.

```
private fun parsePolyline(): Polyline {
        readToken(POLYLINE)
        readToken(LBRACKET)
        val points = parsePoints()
        readToken(RBRACKET)
        readToken(SEMICOLON)
        return Polyline(points)
    }

    private fun parsePolygon(): Polygon {
        readToken(POLYGON)
        readToken(LBRACKET)
        val points = parsePoints()
        readToken(RBRACKET)
        readToken(SEMICOLON)
        return Polygon(points)
    }

    private fun parseCircle(): Circle {
        readToken(CIRCLE)
        readToken(LBRACKET)
        val center = parsePoint()
        readToken(COMMA)
        val radius = parseCircleValue()
        readToken(RBRACKET)
        readToken(SEMICOLON)
        return Circle(center, radius)
    }
```

Funkcije parsePolyline, Polygon in Circle obdelajo seznam točk oz. ene same točke in polmer (za krog).

```
 private fun parseCircleValue(): Expr {
        return when (currentToken.type) {
            FST, SND -> parseFirstSecond()
            NUMBER -> {
                val numberToken = readToken(NUMBER)
                NumberLiteral(numberToken.lexeme.toDouble())
            }
            else -> throw IllegalArgumentException("Expected FST/SND or NUMBER in
circle but got ${currentToken.type}")
        }
    }
```

Funkcija parseCircleValue se uporablja za polmer kroga, podpira pa First in Second koordinato ali pa literal(število).

```
 private fun parsePoints(): List<Point> {
        val points = mutableListOf<Point>()
        points.add(parsePoint())
        while (currentToken.type == COMMA) {
            readToken(COMMA)
```

```
            points.add(parsePoint())
        }
        return points
    }
```

Funkcija parsePoints() sprejme več točk z vejicami, prebere točko in jo doda, nato pride do vejice in po vejici doda še drugo točko.

```
private fun parsePoint(): Point {
        val components = mutableListOf<PointComponent>()
        components.add(parsePointComponent())

        while (currentToken.type == PLUS || currentToken.type == MINUS) {
            val op = readToken(currentToken.type).type
            val next = parsePointComponent()
            components.add(next)
        }

        return Point(components)
    }
```

Funkcija parsePoint() doda vsako komponento točke, ker je točka sestavljena iz več komponent.

```
private fun parsePointComponent(): PointComponent {
        return when (currentToken.type) {
            LPAREN -> {
                readToken(LPAREN)
                val x = parseCoordinate()
                readToken(COMMA)
                val y = parseCoordinate()
                readToken(RPAREN)
                CoordinatePair(x, y)
            }
            DOLLAR -> {
                readToken(DOLLAR)
                val name = readToken(ID).lexeme
                VariableRef(name)
            }
            FST -> {
                readToken(FST)
                val inner = parsePoint()
                First(inner)
            }
            SND -> {
                readToken(SND)
                val inner = parsePoint()
                Second(inner)
            }
            else -> throw IllegalArgumentException("Expected point or variable in
POINT but got ${currentToken.type}")
```

```
        }
    }
```

Funkcija parsePointComponent() prepozna posamezne dele točke, torej: par koordinat (x,y), referenca na spremenljivko ($a), first (fst) ter second (snd).

```
private fun parseCoordinate(): Expr {
        return when (currentToken.type) {
            NUMBER -> {
                val value = readToken(NUMBER).lexeme.toDouble()
                NumberLiteral(value)
            }
            ID -> {
                val name = readToken(ID).lexeme
                VariableReference(name)
            }
            FST -> {
                readToken(FST)
                val inner = parseCoordinate()
                FirstCoordinate(inner)
            }
            SND -> {
                readToken(SND)
                val inner = parseCoordinate()
                SecondCoordinate(inner)
            }
            LPAREN -> {
                readToken(LPAREN)
                val expr = parseCoordinate()
                readToken(RPAREN)
                expr
            }
            else -> throw IllegalArgumentException("Expected coordinate expression but
got ${currentToken.type}, line: ${currentToken.line}, col: ${currentToken.column} ")
        }
    }
```

Funkcija parseCoordinate() se uporablja za obdelovanje posamezne vrednosti x ali y. Lahko je število, spremenljivka, first, second ali zaviti izraz (x).

```
private fun parseFirstSecond(): Expr {
        return when (currentToken.type) {
            FST -> {
                readToken(FST)
                val inner = parseCoordinate()
                FirstCoordinate(inner)
            }
            SND -> {
                readToken(SND)
                val inner = parseCoordinate()
                SecondCoordinate(inner)
```

```
            }
            else -> throw IllegalArgumentException("Expected FST or SND but got
${currentToken.type}")
        }
    }
```

Funkcija parseFirstSecond omogoča, da lahko podamo polmer samo prve (fst) ali druge (snd) komponente nekega izraza.

```
private fun parseVariableDef(): VariableDefinition {
        readToken(LET)
        readToken(AT)
        val name = readToken(ID).lexeme
        readToken(EQUAL)
        val value = parsePoint()
        readToken(SEMICOLON)
        return VariableDefinition(name, value)
    }
```

Funkcija parseVariableDef() obdeluje definicijo nove spremenljivke. Najprej imamo rezervirano besedo let, nato uporabimo @, damo ime spremenljivke, znak "je enako" in damo neko vrednost.

```
private fun parseQuery(): Query {
        readToken(QUESTION_MARK)
        readToken(LBRACE)
        readToken(LBRACKET)
        val points = parsePoints()
        readToken(RBRACKET)
        readToken(COMMA)
        readToken(LBRACKET)
        val center = parsePoint()
        readToken(COMMA)
        val radius = parseCircleValue()
        readToken(RBRACKET)
        readToken(RBRACE)
        readToken(SEMICOLON)
        return Query(points, center, radius)
    }
```

Funkcija parseQuery() obdeluje povpraševanja, ki pa se začnejo z ?, v oglatih oklepajih pa navedemo seznam točk.

9 Implementacija izvoza v GeoJSON

```
fun Expr.toGeoJSON(variables: List<VariableDefinition>): String {
    return when (this) {
        is City -> {
            val features = blocks.map { it.toGeoJSONFeature(variables) }
            """{
                "type": "FeatureCollection",
                "features": [${features.joinToString(",")}]
```

```
            }"""
        }
        else -> throw IllegalArgumentException("Unsupported Expr type for GeoJSON
conversion")
    }
}
```

Najprej imamo glavno funkcijo toGeoJSON(), ki preveri ali je izraz instance City, če je potem pretvori vsak blok v GeoJSON "feature", združi pa jih v "FeatureCollection", kar je standardni format GeoJSON-a za več objektov. Če ni City, potem vrže napako.

```
fun Block.toGeoJSONFeature(variables: List<VariableDefinition>): String {
    return when (this) {
        is Road -> """{
            "type": "Feature",
            "geometry": {
                "type": "LineString",
                "coordinates": ${polyline.points.toGeoJSONCoordinates(variables)}
            },
            "properties": {"name": "$name"}
        }"""
        is Building -> """{
            "type": "Feature",
            "geometry": {
                "type": "Polygon",
                "coordinates": [${(shape as
Polygon).points.toGeoJSONCoordinates(variables)}]
            },
            "properties": {"name": "$name"}
        }"""
        is Area -> """{
            "type": "Feature",
            "geometry": {
                "type": "Polygon",
                "coordinates": [${(shape as
Polygon).points.toGeoJSONCoordinates(variables)}]
            },
            "properties": {"name": "$name"}
        }"""
        is Park -> """{
            "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates": ${(shape as
Circle).center.toGeoJSONCoordinates(variables)},
                "radius": ${(shape as Circle).radius.toGeoJSONValue()}
            },
            "properties": {"name": "$name"}
        }"""
        is Lake -> """{
            "type": "Feature",
```

```
            "geometry": {
                "type": "Point",
                "coordinates": ${(shape as
Circle).center.toGeoJSONCoordinates(variables)},
                "radius": ${(shape as Circle).radius.toGeoJSONValue()}
            },
            "properties": {"name": "$name"}
        }"""
        else -> throw IllegalArgumentException("Unsupported Block type for GeoJSON
conversion")
    }
}
```

Funkcija Block.toGeoJSONFeature() pretvori vsak blok v posamezen GeoJSON "feature". Cesta je LineString, s pomočjo polyline, zgradba ali območje je poligon. Park ter jezero je točka, ki ima polmer.

```
fun List<Point>.toGeoJSONCoordinates(variables: List<VariableDefinition>): String {
    val nestedCoordinates = map { it.toGeoJSONCoordinates(variables) }
    return "[${nestedCoordinates.joinToString(",")}]"
}
```

Funkcija List<Point>.toGeoJSONCoordinates() za seznam točk ustvari seznam koordinat ter lepo oblikuje z oglatimi oklepaji.

```
fun Point.toGeoJSONCoordinates(variables: List<VariableDefinition>): String {
    val pair = components.filterIsInstance<CoordinatePair>().firstOrNull()
    if (pair != null) {
        return "[${pair.x.toGeoJSONValue()}, ${pair.y.toGeoJSONValue()}]"
    }

    val variableRef = components.filterIsInstance<VariableRef>().firstOrNull()
    if (variableRef != null) {
        val resolvedPoint = resolveVariableRef(variableRef, variables)
        return resolvedPoint.toGeoJSONCoordinates(variables)
    }

    throw IllegalArgumentException("Point must contain a CoordinatePair or a
resolvable VariableRef.")
}
```

Funkcija Point.toGeoJSONCoordinates() prevaja Point v GeoJSON koordinato. Če vsebuje par koordinat, vrne [x,y], če vsebuje referenco na spremenljivko, potem poskusi poiskati točko v seznamu spremenljivk in tudi pretvori. Če ne najde ničesar, vrne napako.

```
fun Expr.toGeoJSONValue(): String {
    return when (this) {
        is NumberLiteral -> value.toString()
        else -> throw IllegalArgumentException("Unsupported Expr type for GeoJSON
value conversion")
    }
}
```

Funkcija Expr.toGeoJSONValue() pretvori številske izraze v tekstovno obliko za GeoJSON.

```
fun resolveVariableRef(ref: VariableRef, variables: List<VariableDefinition>): Point {
    val definition = variables.find { it.name == ref.name }
        ?: throw IllegalArgumentException("VariableRef '${ref.name}' is not defined.")
    return definition.value
}
```

Funkcija resolveVariableRef() poišče definicijo spremenljivke z istim imenom v seznamu spremenljivk. Če jo najde, potem vrne njeno vrednost (točko), če ne vrne napako.

10 Konstrukcija končnega avtomata

Končni avtomat je uporabljen znotraj pregledovalnika, bolj podrobno v funkciji scanTokens(), kjer vsak znak iz niza `input` predstavlja vhodni simbol avtomata, trenutni znak (ch) določa, katero stanje se aktivira. Določena zaporedja znakov povzročijo prehod v končno stanje, kjer se prepozna in shrani token.

```
in '0'..'9' -> {
    val startColumn = column
    val number = readNumber()
    tokens.add(Token(TokenType.NUMBER, number, line, startColumn))
}
```

Prepoznavanje števil: Najprej preberemo znak `ch`, če je znak številka, potem gremo to brati kot število. Ko pridemo do konca, dodamo žeton v seznam.

```
if (ch.isLetter() || ch == '_') {
    val startColumn = column
    val word = readWhile { it.isLetterOrDigit() || it == '_' }
    val type = when (word) {
        "city" -> TokenType.CITY
        "road" -> TokenType.ROAD
        ...
        else -> TokenType.ID
    }
    tokens.add(Token(type, word, line, startColumn))
}
```

Prepoznavanje ključnih besed in spremenljivk: Najprej preberemo znak `ch`, če je črka, potem gremo tako daleč dokler ne pridemo do konca besede. Prepoznamo, ali je beseda ena izmed ključnih, če ni potem smatramo da je ime spremenljivke. Na koncu dodamo žeton v naš seznam.

```
'"' -> {
    val startColumn = column
    advance() // skip "
    val str = readWhile { it != '"' }
    advance() // skip closing "
    tokens.add(Token(TokenType.STRING, str, line, startColumn))
}
```

Prepoznavanje nizov: Znova preberemo znak `ch` in če je to narekovaj, potem se premaknemo eno naprej in beremo dokler ne pridemo do zaključnega narekovaja. To kar je med njima, je naš niz, ki ga dodamo kot žeton.

```
'{' -> addToken(TokenType.LBRACE, "{")
'}' -> addToken(TokenType.RBRACE, "}")
'[' -> addToken(TokenType.LBRACKET, "[")
']' -> addToken(TokenType.RBRACKET, "]")
'(' -> addToken(TokenType.LPAREN, "(")
')' -> addToken(TokenType.RPAREN, ")")
',' -> addToken(TokenType.COMMA, ",")
';' -> addToken(TokenType.SEMICOLON, ";")
'=' -> addToken(TokenType.EQUAL, "=")
'+' -> addToken(TokenType.PLUS, "+")
'-' -> addToken(TokenType.MINUS, "-")
'?' -> addToken(TokenType.QUESTION_MARK, "?")
'@' -> addToken(TokenType.AT, "@")
'$' -> addToken(TokenType.DOLLAR, "$")
```

Za enoznakovne simbole je bolj preprosto: Preberemo znak `ch` in glede na to kar najdemo, tudi dodamo kot žeton v naš seznam.

11 Priprava smiselnih testnih primerov

11.1 Primer

```
city{
    road["Ptujska cesta"]{
        polyline[(1,2),(3,4),(5,6)];
    };
    area["FERI"]{
        polygon[(1,1),(1,2),(2,2),(2,1),(1,1)];
    };
    area["Park"]{
        circle[(4,3),2];
    };
    let $p = (1,1);
    let $q = (2,2);

    road["Ljubljanska cesta"]{
        polyline[$p,$q,$p+$q];
    };

    ?{[(1,1),$p,$q,(3,4)],[(1,1),3]};

    let $r = (fst(p),snd(q));
```

11.2 Primer

```
let $a = (0,0);
let $b = (1,1);
let $c = (2,2);
```

```
let $d = (3,3);

city["Veliko mesto"] {
    road["Glavna ulica"] {
        polyline[$a,$b,$c,$d];
    };

    building["Občina"] {
        polygon[(1,1),(2,1),(2,2),(1,2),(1,1)];
    };

    area["Trg"] {
        polygon[(3,3),(4,3),(4,4),(3,4),(3,3)];
    };

    park["Zeleni park"] {
        circle[(5,5),2.5];
    };

    lake["Veliko jezero"] {
        circle[(6,6),4];
    };
}
```

11.3 Primer

```
let $x = (3,3);
let $y = (6,6);

city["Kompleksno mesto"] {
    road["Severna"] {
        polyline[(0,0),$x,(6,0)];
    };
    building["Muzej"] {
        polygon[(2,2),(4,2),(4,4),(2,4),(2,2)];
    };
    area["Stadion"] {
        polygon[(5,5),(6,6),(7,5),(6,4),(5,5)];
    };
    park["Jugozahodni park"] {
        circle[(1,1),2];
    };
    lake["Ribnik"] {
        circle[$y, 1.5];
    };
    road["Vzhodna"] {
        polyline[$x, $y, $x+$y];
    };

    ?{[(3,3),$x,$y,(5,5)],[(4,4),2]};
}
```

11.4 Primer

```
let $a = (1,1);
let $b = (2,3);
let $c = $a + $b;
let $d = (fst($b), snd($c));

city["Ljubljana"]{
    road["Dunajska"]{
        polyline[(1,2), (2,3), $c];
    };
    road["Celovška"]{
        polyline[$a, (2,2), $d];
    };
    building["Modra stavba"]{
        polygon[(1,1),(1,2),(2,2),(2,1),(1,1)];
    };
    area["Trg Republike"]{
        polygon[(1,1), (2,4), (4,4), (4,2), (1,1)];
    };
    lake["Zbiljsko jezero"]{
        circle[(5,5), 2.5];
    };
    park["Park Tivoli"]{
        circle[(4,3), 3];
    };
}

?{[$a, $b, $c, $d],[(2,2),2]};
let $rez = ($a + $b) - (1,1);
```

11.5 Primer

```
let $p1 = (3,3);
let $p2 = (1,2);
let $p3 = (2,2);
let $mid = ($p1 + $p2) - (1,1);
let $origin = (0,0);
let $z = (fst($mid), snd($p3));

city["Koper"]{
    road["Obalna"]{
        polyline[$p1, $p2, $p3, $p1];
    };
    area["Trg Koper"]{
        polygon[(1,1), (2,2), (2,4), (1,1)];
    };
    building["Zgradba A"]{
        polygon[(0,0), (0,2), (2,2), (2,0), (0,0)];
    };
    lake["Jezero Koper"]{
        circle[$p2, 1.8];
```

```
    };
}

city["Celje"]{
    road["Kidričeva"]{
        polyline[(0,0), $mid, $mid + (1,1)];
    };
    park["Mestni park"]{
        circle[$origin, 3];
    };
    area["Stari trg"]{
        polygon[(1,1),(2,2),(3,3),(4,4),(1,1)];  // validacija: pravilno zaprt
    };
    building["Dvorana"]{
        polygon[(1,1),(1,3),(2,3),(2,1),(1,1)];
    };
}

?{[ $p1, $p2, $p3, (4,4) ], [ $p2, 2.0 ]};
let $v = (fst($p1) + 1, snd($p2) - 1);
```