

Algoritmer för syntaxanalys

OH-serie 3: Tillståndsmaskiner, recursive descent och implementation därav i Java

Mats Dahllöf
Institutionen för lingvistik och filologi
Maj 2008.



UPPSALA
UNIVERSITET

1

Algoritmer för syntaxanalys — VT 2008 (Mats Dahllöf)

Tillståndsmaskin

En grammatik, input och trädbyggnadsstrategi kan förstås som en tillståndsmaskin, med:

- Definition av initialkonfiguration.
- Definition av acceptansvillkor.
- Definition av övergångar:
 - villkor som säger vilka som är tillåtna
 - definition av hur den konfiguration ser ut som man når genom en övergång

2

Algoritmer för syntaxanalys — VT 2008 (Mats Dahllöf)

En exempelimplementation i Java

- Separation tillståndsmaskin och sökmekanism (backtracking).
- Gemensamma klasser för flera olika algoritmer för (parsning och) igenkänning ("recognition").
- Detta kräver viss överflödlig information. (Implementationen kan rensas om man bara vill ha en algoritm.)

3

Algoritmer för syntaxanalys — VT 2008 (Mats Dahllöf)

Översikt, klasser, för Recursive Descent

- CFG, CFGlexEntry och CFGrule för grammatik och grammatiksatser (genrella, för de implementerade algoritmerna).
- Config(uration) för konfigurationer (generell).
- Transit(ion) för övergångar (generell).
- RecursiveDescent för definition av tillståndsmaskinen för recursive descent (som motsvarar trädkonstruktion top-down, djupet-först).

4

Algoritmer för syntaxanalys — VT 2008 (Mats Dahllöf)

Översikt, klasser, för Recursive Descent

- (StateMachine gränssnitt för tillståndsmaskinerna.)
- Backtracker för sökmekanismen (backtracking, djupet-först) (generell).
- ParserRD för att testa RecursiveDescent.
- TreeEdge för att samla ihop trädinformation (används nu ej av RecursiveDescent, man får bara en tom lista av TreeEdge's) (generell).
- Feedback för spåringsutskriften (generell).

5

Algoritmer för syntaxanalys — VT 2008 (Mats Dahllöf)

Konfigurationer

```
public class Config {
    // Nr of Config's created.
    private static int counter = 0;
    // Config identifier
    private int id = 0;
    // Reading position
    private int readPos = 0;
    // Stack (of Node's)
    private Node[] stack = new Node[0];
}
```

6

Algoritmer för syntaxanalys — VT 2008 (Mats Dahllöf)

Noderna (på stacken)

```
public class Node {
    // For the category.
    private String symbol = "";
    // Defines whether the node is verified ("+")
    // or predicted ("-") (for Left corner parsing)
    private String sign = "";
    // To identify a Node (e.g. in syntax trees)
    private int id = 0;
}
(Endast symbol viktig vid Rec. desc.)
```

7

Algoritmer för syntaxanalys — VT 2008 (Mats Dahllöf)

Övergångarna

```
public class Transit {
    // Identifies the kind of transition.
    // E.g. "expand" or "scan" in Rec. desc.
    private String transType = "";
    // For category shift or dependency labels
    private String label = "";
    // The rule with expand or reduce.
    private CFGrule rule = new CFGrule();
}
(label används ej vid Rec. desc.)
```

8

RecursiveDescent's instansvariabler

```
public class RecursiveDescent
    implements StateMachine {
    private CFG grammar;
    private String[] input;
    StateMachine: ett gränssnitt mot tillståndsmaskinerna, varav
    RecursiveDescent är en.
```

9

En StateMachine's metoder (generellt)

- public Config initialConfig()
- public boolean accepting(Config c)
- public LinkedList<Transit> findTransits(Config c)
Ger en lista på tillåtna övergångar. (Kollar villkor.)
- public Config computeTransit(Config c, Transit t)
Räknar ut vilken konfiguration man når vid en övergång från en annan. (Utför.)

10

Initialkonfiguration, RecursiveDescent

```
public Config initialConfig() {
    Config config = new Config();
    config.pushStack(new Node("S", "-"));
    config.setReadPos(0);
    return config;
}
```

Annan notation, $\langle\langle S \rangle, 0\rangle$.

11

Acceptans, RecursiveDescent

```
public boolean accepting(Config config) {
    return (config.stackDepth() == 0 &&
            config.readPos() == input.length);
}
```

Annan notation, $\langle\langle \rangle, l\rangle$.

12

findTransits, RecursiveDescent

Hitta expansioner (kodsnitt):

```
public LinkedList<Transit>
    findTransits(Config config) {
    [...]
    LinkedList<CFGrule> applicableRules =
        grammar.rulesExpandingCat(
            config.peekStack(0).cat());
    for (CFGrule rule: applicableRules) {
        transits.add(new Transit("exp",rule));};
```

13

findTransits, RecursiveDescent

Hitta scan-övergångar (kodsnitt):

```
public LinkedList<Transit>
    findTransits(Config config) {
    [...]
    if (config.readPos() < input.length &&
        grammar.isOfCat(input[config.readPos()],
            config.peekStack(0).cat())) {
        transits.add(new Transit("scan",""));}
```

14

computeTransit, RecursiveDescent

```
public Config
    computeTransit(Config c, Transit t) {
    if (t.ofType("exp")) {
        return expand(c,t.rule());
    }
    else if (t.ofType("scan")) {
        return scan(c);
    }; [...]
```

Kopplar Transit-objekt till "utföremetoder".

15

"Utföremetod", expansion

```
private Config expand(Config c, CFGrule rule) {
    Config newc = new Config();
    // kopiera stack, poppa en nod
    newc.copyPoppedStack(c,1);
    // pusha prediktioner från regeln
    newc.addPredictions(rule,0);
    // samma läsposition:
    newc.setReadPos(c.readPos());
    return newc;};
```

16

Jämför expansion

Grammatik: $\langle T, N, P, S \rangle$. Input: t_0, \dots, t_{l-1} .

Expansion utifrån $C_0 \rightarrow R_0 \dots R_m$

- *Konfiguration*: $\langle \langle C_0, \dots, C_n \rangle, i \rangle$.
- *Villkor*: Regeln $C_0 \rightarrow R_0 \dots R_m$ finns i P .
- *Ny konfiguration*: $\langle \langle R_0, \dots, R_m, C_1, \dots, C_n \rangle, i \rangle$.

17

"Utförarmetod", scanning

```
private Config scan(Config c) {
    Config newc = new Config();
    // kopiera stack, poppa en nod
    newc.copyPoppedStack(c, 1);
    // gå framåt ett steg i läsning
    newc.setReadPos(c.readPos() + 1);
    return newc;}

```

18

Jämför scanning

Grammatik: $\langle T, N, P, S \rangle$. Input: t_0, \dots, t_l .

Scanning

- *Konfiguration*: $\langle \langle C_0, \dots, C_n \rangle, i \rangle$.
- *Villkor*: Det finns en lexikoningång $C_0 \rightarrow t$ i P och $t = t_i$.
- *Ny konfiguration*: $\langle \langle C_1, \dots, C_n \rangle, i + 1 \rangle$.

(Vi tar ett steg framåt i läsningen av input och plockar bort stackens översta icke-terminal. Villkoret är terminalen är av icke-terminalens kategori enligt en lexikoningång.)

19

Backtracker, instansvariabler

Kan, givet en StateMachine, göra en sökning djupet-först med backtracking från intalkonfiguration till acceptanskonfigurationer.

```
public class Backtracker {
    private StateMachine machine;
    // Stack för konfigurationer + möjliga transit.
    private LinkedList<TrackerState> states = new
        LinkedList<TrackerState>();
    private int transitionsFollowed = 0;
}

```

20

Backtracker, en intern klass

TrackerState är en intern klass:

```
public class Backtracker {
    [...]
    private class TrackerState {
        public Config config;
        // For not yet explored transitions
        // from config:
        public LinkedList<Transit> transits;
    }
}

```

21

Backtracker's centrala metod

```
public LinkedList<LinkedList<TreeEdge>> parse()

```

Runs the state machine on the current input by following all possible transitions depth first, by backtracking, and as ordered by the machine.

Returns the list of syntax trees (each tree being a `LinkedList<TreeEdge>` object) associated with the acceptance configurations that are reached during the search process.

(The trees are empty for the recognizer `RecursiveDescent`.)

22