Grundlagen der Wissensverarbeitung – Tutorial 5, Gruppe 4

Arne Beer MN 6489196 Marta Nevermann MN 6419716
Daniel Waller MN 6813853 Julius Hansen MN 6455291

Exercise 1.2

1.

(a) Es gibt einen Stack S, eine Inputliste I und eine Relation A. Folgende Operationen werden verwendet, um den gewünschten Output zu generieren: `Left-Arc` generiert eine Kante vom nächsten Element in I zum obersten Element in S und entfernt (pop) es gleichzeitig vom Stack. `Right-Arc` erzeugt eine Kante vom obersten Element auf S zum nächsten Inputtoken und schiebt letzteres auf den Stack. `Reduce` entfernt lediglich das oberste Element auf dem Stack und `Shift` legt das nächste Inputelement auf den Stack.

(b) Der Parser terminiert in dem Zustand [S, nil, A], mit einer beliebigen Liste S, einer leeren Inputliste und einer Grammatik, also einem gerichteten Abhängigkeitsgraphen A. Damit der entstandene Graph akzeptiert wird, muss er bestimmte Bedingungen erfüllen. (siehe (c))

(c) Ein generierter Abhängigkeitsgraph muss bestimmte Abhängigkeiten erfüllen, um akzeptiert zu werden: Er ist azyklisch, single-headed (jeder Knoten ist höchstens von einem anderen Knoten abhängig), verbunden (entspricht einem schwach zusammenhängendem Graphen) und projektiv (jeder abhängige Knoten ist adjazent zu seinem Kopf, das heißt alle Knoten, die zwischen ihnen liegen sind von einem der Beiden abhängig).

(d) Graphen als PDF Files extra, transitive Relationen bitte nicht beachten

2.

`Shift` S
`Left-Arc` LA

`Right-Arc` RA

Startzustand: [nil, der Mann isst eine Giraffe, {0}]
`S` -> [der, Mann isst eine Giraffe, {0}]
`LA` -> [nil, Mann isst eine Giraffe, {(Mann, der)}]
`S` -> [Mann, isst eine Giraffe, {(Mann, der)}]
`LA` -> [nil, isst eine Giraffe, {(Mann, der), (isst, Mann)}]
`S` -> [isst, eine Giraffe, {(Mann, der), (isst, Mann)}]
`S` -> [eine isst, Giraffe, {(Mann, der), (isst, Mann)}]
`LA` -> [isst, Giraffe, {(Mann, der), (isst, Mann), (Giraffe, eine)}]
`RA` -> [Giraffe isst, nil, {(Mann, der), (isst, Mann), (Giraffe, eine), (isst, Giraffe}]
(Endzustand)

3.

1. Die `search states` des Parsing Algorithmus sind die jeweilig möglichen Zustände, die das Tripel (S,I,A) annehmen kann.

2. Der `start state` ist der Zustand, in dem der noch vollständige Satz in der Inputliste liegt, der Stack leer und die Relation die leere Menge ist, also (nil, I, {}).

3. Die `end states` sind alle Zustände, bei denen die Inputliste leer ist und ein Stack S und eine Relation A existiert, also (S,nil,A)

4. Die `state transitions` sind alle definierten zulässigen Operationen auf dem Tripel, folglich: Right-Arc, Left-Arc, Reduce, Shift.

5. Nein. Der `search space` wird bei diesem Algorithmus erst durch die Anwendung der Operationen entdeckt. Zu Beginn ist lediglich der `start state` (nil, I, {}) bekannt, von dem der mögliche `search space` erkundet wird.

6. Der Algortithmus läuft mit linearer Komplexität und terminiert garantiert, so dass alleine das generieren aller möglichen Bäume mehr Zeit in Anspruch nehmen würde.

7. Alle Algorithmen, welche sich anhand von Kosten orientieren oder mithilfe einer Heuristik, sind in diesem Fall nutzlos, da hier kein klares Kostenmodell aufgestellt werden kann.