5.7 Difference Lists and Definite Clause Grammars

Freitag, 26. Juni 2015 08:30

Goal: • Parsing (i.e., solving the word problem for context-free languages).

Solution: • Prolog offers special support for context-free grammars
• Efficient because of the use of <u>difference lists</u>.

5.7.1. Difference Lists

Goal: more efficient implementation of list operations.

Ex: app/3 for list concatenation

?- app([1,2,3], [4,5], Zs).
Zs = [1,2,3,4,5]

Complexity: O(n) where n is the length of the list in the first argument.

Goal: find an alternative append-implementation with complexity O(1).

Idea: use a different representation of lists: <u>Difference Lists</u>

Difference Lists

[1,2,3] can be represented as [1,2,3,4,5] - [4,5]

Representation is not unique.
[1,2,3] could also be represented as

[1,2,3,4,5 | Ys] - [4,5 | Ys] or

[1,2,3 | Ys] - Ys etc.

(most general difference list representing [1,2,3])

Alternative implementation of app:

app(Xs-Ys, Ys, Xs).

?- app([1,2,3|Ys]-Ys, [4,5], Zs).

Zs = [1, 2, 3, 4, 5]

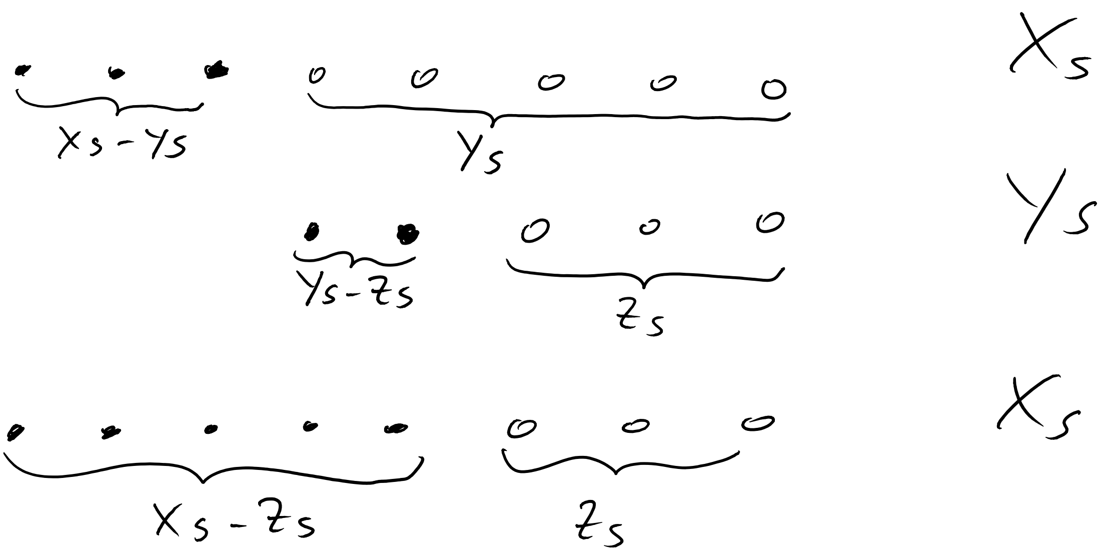
(- is not related to pre-defined subtraction. One could also any other fct. symbol.)

Reason: in 1 resolution step we obtain □
using mgu: Ys = [4,5],
Xs = [1,2,3,4,5]
Zs = —— " ——

Disadvantage: only arg 1 is in difference list-representation. ⇒ app cannot be used repeatedly.

Better version, where all arguments of app are difference lists:

app(Xs-Ys, Ys-Zs, Xs-Zs).



app(Xs-Ys, Ys-Zs, Xs-Zs).

?- app([1,2,3|Ys]-Ys, [4,5|Zs]-Zs, Res).

Ys = [4,5|Zs]

Xs = [1,2,3,4,5|Zs]

Res = [1,2,3,4,5|Zs] - Zs

Now we obtain the result in difference list-repres.

Computation only needs 1 resolution step (O(1)).

app(Xs-Ys, Ys-Zs, Xs-Zs)

only works if the first 2 arguments are represented in a "compatible" way.

e.g. :?- app([1,2,3,6]-[6], [4,5]-[ ], Res).

do not unify

false

Better: use the most general difference list representation (e.g. [1,2,3|Ys]-Ys).

5.7.2. Definite Clause Grammars

Prolog allows representation of context-free grammars and it directly contains an efficient algorithm for parsing, based on difference lists.

→ Parsers for different languages can be easily implemented in Prolog.

Context-free grammar:

$G = (N, T, S, P)$ where

N: set of non-terminals

T: set of terminals

S: $S \in N$ start symbol

P: Set of productions (rules) of the form:

$$A \to \alpha \quad \text{with } A \in N, \alpha \in (N \cup T)^*$$

G defines a derivation relation $\Rightarrow_G$ between words:

$$\beta \Rightarrow_G \gamma \quad \text{iff}$$

there is a $A \to \alpha \in P$ such that

$\beta = \beta_1 \, A \, \beta_2$ and

$\gamma = \beta_1 \, \alpha \, \beta_2$

Grammar G defines the language

$L(G) = \{ w \in T^* \mid S \Rightarrow_G^* w \}$.

Ex: Sentence $\Rightarrow_G$

Nominalphr Verbalphr $\Rightarrow_G$

Article Noun Verbalphr $\Rightarrow_G$

a Noun Verbalphr $\Rightarrow_G$ ...

a cat scares the mouse

Representation of context-free grammars in Prolog:

- Non-terminals of N are written as constants (i.e, as predicate symbols of arity 0).
- Terminals of T are written singleton lists with a constant (e.g., [cat]).
- Words of T^* are written as lists of constants (e.g. [a, mouse, hates]). The empty word ε is written as [].
- Words of $(N \cup T)^*$ are written as sequences of constants and lists of constants. So "a mouse Verb Nominalphrase" is written as "[a, mouse], verb, nominalphrase".
- Instead of "->", one writes -->.

Prolog translates rules built with --> into ordinary clauses.

First idea for such a translation:

- Every non-terminal could correspond to a unary predicate which checks whether its argument can be derived from this non-terminal.
- a --> [a1, a2, a3] would be translated to the clause:
  (a: non-terminal; a1, a2, a3: terminals)

  a([a1, a2, a3]). ← states that the word $a_1 a_2 a_3$ can be derived from a.

  Ex: verb --> [scares] would be translated to

  verb([scares]).
- a --> a1 would be translated to

  a(A) :- a1(A).

  Ex: verbalphrase --> verb would be transl. to

  verbalphrase(A) :- verb(A).
- a --> a1, a2 would be translated into

  a(A) :- append(A1, A2, A),
  a1(A1).

$a_2(A_2)$.

Ex: sentence --> nominalphr, verbalphr. is translated to

sentence(S) :- append(NP, VP, S),
nominalphr(NP), verbalphr(VP).

Drawback: inefficient, because append is called repeatedly (due to backtracking).

Solution: use difference lists instead.

Then: a(A-B) would hold iff

from the non-terminal a one can derive the word A without its suffix B.

Prolog uses a representation of difference lists with 2 arguments: a(A,B) instead of a(A-B).

⇒ For every non-terminal a, Prolog creates a predicate symbol a/2.

a(A,B) holds iff from a one can derive the word/list A without its end B.

- a --> a_1 is translated to

$a(A, B) :- a_1(A, B)$.

- a --> a1, a2 is translated to

$$a(A,B) \text{ :- } app(X_s\text{-}Y_s,\ V_s\text{-}W_s,\ A\text{-}B),\ a_1(X_s, Y_s),\ a_2(V_s, W_s).$$

Alternative more elegant formulation:

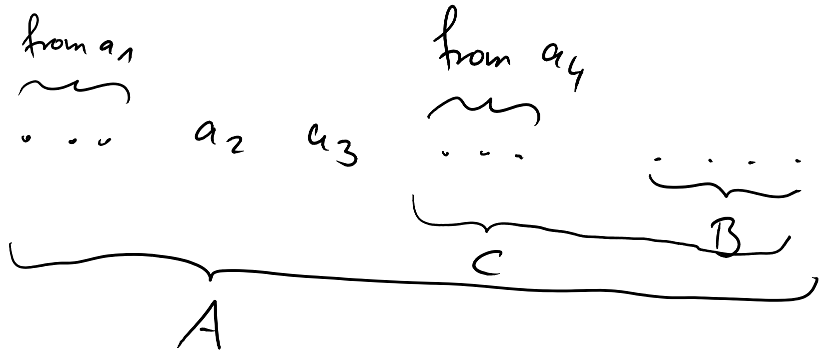
$$a(A,B) \text{ :- } a_1(A, C),\ a_2(C, B).$$

- a --> [a1, a2, a3] is translated into

$$a([a_1, a_2, a_3 \mid X_s], X_s).$$

- a --> a1, [a2, a3], a4 is translated into

$$a(A,B) \text{ :- } a_1(A, [a_2, a_3 \mid C]),\ a_4(C, B).$$



Use of this prog. for parsing:

```
?- sentence([the, cat, scares, a, mouse], []).
true
?- sentence([the, cat, scares, a, mouse, trash], [trash]).
true
```

```
?- sentence(S, []).
S = [a, cat, scares] ;
S = [a, cat, hates] ;
  ⋮
```