5.7 Difference Lists and Definite Clause Grammars

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 difference lists.

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: Difference Lists
Difference Lists

\[ [1,2,3] \text{ can be represented as } [1,2,3,4,5] - [4,5] \]

Representation is not unique.
\[ [1,2,3] \text{ could also be represented as } \]
\[ [1,2,3,4,5 | Ys] - [4,5 | Ys] \text{ or } \]
\[ [1,2,3 | Ys] - Ys \]
\[ \text{etc.} \]

Alternative implementation of \text{app}:
\[
\text{app} \left( Xs - Ys, Ys, Xs \right).
\]

\[ ? - \text{app} \left( [1,2,3 | Ys] - Ys, [4,5], Zs \right). \]
\[ Zs = [1,2,3,4,5] \]

Reason: in \text{1} resolution step we obtain \[ \square \]

using \text{mgu}: \[ Ys = [4,5], \]
\[ Xs = [1,2,3,4,5] \]
\[ Zs = \square \]

Disadvantage: only \text{arg} \text{1} \text{is in difference list representation}. \text{app} cannot be reused repeatedly.

Better version, where all arguments of \text{app} are difference lists:
\( \text{app}(X_5 - Y_5, Y_5 - Z_5, X_5 - Z_5) \).

\[-\text{app}(X_5, Y_5, Z_5).\]

\( \frac{\text{app}(X_5 - Y_5, Y_5 - Z_5, X_5 - Z_5)}{\text{app}(\lfloor 1, 2, 3 \rfloor Y_5 - Y_5, \lfloor 4, 5 \rfloor Z_5 - Z_5, \text{Res}).} \)

\( Y_5 = \lfloor 4, 5 \rfloor Z_5 \)
\( X_5 = \lfloor 1, 2, 3, 4, 5 \rfloor Z_5 \)
\( \text{Res} = \lfloor 1, 2, 3, 4, 5 \rfloor Z_5 - Z_5 \)

Now we obtain the result in difference list - representation.
Computation only needs 1 resolution step (O(n)).

\( \text{app}(X_5 - Y_5, Y_5 - Z_5, X_5 - Z_5) \)
only works if the first 2 arguments are represented in a "compatible" way.
e.g. \( \text{app } ([1,2,3,6], [6], [4,5,7], [7], \text{Res}) \).

\[\text{false}\]

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.

\(\rightarrow\) 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
- \(S\) : start symbol
- \(P\) : set of productions (rules) of the form:
  \[A \rightarrow \alpha\quad \text{with } A \in N, \alpha \in (N \cup T)^*\]

\(G\) defines a derivation relation \(\Rightarrow_g\) between words.

\[\beta \Rightarrow_g \gamma \quad \text{iff}\]
there is an $A \rightarrow \alpha \in P$ such that
$B = B_1 A B_2$ and
$S = B_1 \alpha B_2$

Grammar $G$ defines the language
$L(G) = \{ w \in T^* \mid S \Rightarrow^*_G w \}$.

Ex:  **Sentence** $\Rightarrow G$

  **Nominalphrase** Verbphrase $\Rightarrow G$

  **Article** Noun Verbphrase $\Rightarrow G$

  a Noun Verbphrase $\Rightarrow G$ ...

  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 $\epsilon$ 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 "$\Rightarrow$", one writes "$\rightarrow$".
Prolog translates rules built with $\rightarrow$ 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.

- $a \rightarrow [a_1, a_2, a_3]$ would be translated to the clause:

  \[
  a([a_1, a_2, a_3], \text{states that the word } a_1 a_2 a_3 \text{ can be derived from } a).
  \]

  Ex: Verb $\rightarrow$ [escapes] would be translated to

  \[
  \text{verb}([\text{escapes}]).
  \]

- $a \rightarrow a_1$ would be translated to

  \[
  a(A) : - a_1(A).
  \]

  Ex: Verbalphrase $\rightarrow$ verb would be tranl. to

  \[
  \text{verbalphrase}(A) : - \text{verb}(A).
  \]

- $a \rightarrow a_1, a_2$ would be translated into

  \[
  a(A) : - \text{append}(A_1, A_2, A),
  a_1(A_1).
  \]
\[ a_2 (A_2). \]

Example: sentence \( \rightarrow \) nominalphr, verbalphr. is translated to
\[
\text{Sentence } (S) : = \text{append}(\text{NP}, \text{VP}, S), \\
\text{nominalphr}(\text{NP}), \text{verbalphr}(\text{VP}).
\]

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 \( a(A-B) \).

\[ \Rightarrow \text{For every non-terminal } a, \text{ Prolog creates a predicate symbol } a/2. \]

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

\[ a \rightarrow a_n \] is translated to
\[
a(A,B) : = a_n(A,B).
\]
\(a \rightarrow a_1, a_2\) is translated to

\[
a(A, B) : = \text{app}(X_5, Y_5, V_5, W_5, A - B), \\
a_1(X_5, Y_5), \\
a_2(V_5, W_5).
\]

Alternative more elegant formulation:

\[
a(A, B) : = a_1(A, C), a_2(C, B).
\]

\(a \rightarrow [a_1, a_2, a_3]\) is translated into

\[
a([a_1, a_2, a_3], X_5), X_5).
\]

\(a \rightarrow a_1, [a_2, a_3], a_4\) is translated into

\[
a(A, B) : = a_1(A, [a_2, a_3 | C]), a_4(C, B).
\]

Use of this program for parsing:

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

? - sentence ([the, cat, scares, a, mouse, trash], [trav]).
true
?- sentence (S, EI).
S = [a, cat, scares] ;
S = [a, cat, hates] ;
...