Grammatiken in Prolog

Allgemeines:

- Gedacht zur Verarbeitung natürlicher Sprache.

  Dort braucht man kompliziertere Grammatiken als etwa im Compilerbau, andererseits sind die Eingaben meistens kurz, die Effizienz-Anforderungen sind also nicht so hoch.

- Beliebige kontextfreie Grammatiken darstellbar.

  Dagegen können mit Parsergeneratoren wie „yacc“ nur eingeschränkte Grammatiken (in diesem Fall LALR(1)) verarbeitet werden.

- Auch nichtdeterministische Grammatiken.

  Bei denen also eine Eingabe mehrere Abteilungsbäume hat. Gerade in der natürlichen Sprache gibt es ja mehrdeutige Sätze.

- Beliebige Mischung mit Prolog-Code möglich.

  Es können also auch kontextsensitive Bedingungen abgefragt werden. Bei „yacc“ kann man zusätzliche Prüfungen mit C-Code vornehmen, aber diese Prüfung ist von der eigentlichen Grammatik klar getrennt. Es kann also z.B. nicht zur Laufzeit eine Produktion ausgewählt werden.

- „Definite Clause Grammars“
- Realisiert als Präprozessor für Prolog.

  Also keine prizipiell neuen Sprachkonstrukte, sondern nur ein neuer Algorithmus (den man auch ohne den Präprozessor nutzen kann).

- Nicht in Turbo-Prolog und kleinen PD-Prologs.

Syntax (informell)

Beispiel (Adventure-Spiel):

- Kommandos z.B.: „Nimm das Schwert".
- kommando `-->` verb objekt.

  verb `-->` [nimm].

  verb `-->` [reibe].

  objekt `-->` artikel nomen.

  artikel `-->` [].

  artikel `-->` [den].

  artikel `-->` [die].

  artikel `-->` [das].

  nomen `-->` [schwert].

  nomen `-->` [lampe].
- ?- phrase(kommando, [nimm, das, schwert]). yes.

Allgemein:

- Nichtterminalsymbole: Prolog-Atome.
- Terminalsymbole: Listen.
- Analyse mit „phrase(Nichtterminal, Eingabe)".

Implementierung

Übersetzung nach Prolog:

- Idee: Übersetze Nichtterminalzeichen in Prädikate, die für die ableitbaren Folgen von Terminalzeichen gelten, z.B. „objekt([das, schwert])“.
- objekt(Z) :- append(X,Y,Z),artikel(X),nomen(Y).
- So sehr ineffizient! Verbesserung: Differenzlisten, z.B. „artikel([das, schwert], [schwert])“.
- Jedes Prädikat schneidet vorne den Teil ab, den es verarbeiten kann, und reicht den Rest weiter.
- objekt(S0, S2) :- artikel(S0, S1), nomen(S1, S2).

  artikel(S0, S1) :- S0 = S1.
  artikel(S0, S1) :- S0 = [den|S1].
  artikel(S0, S1) :- S0 = [die|S1].
  artikel(S0, S1) :- S0 = [das|S1].

  nomen(S0, S1) :- S0=[schwert|S1].
  nomen(S0, S1) :- S0=[lampe|S1].
- phrase(Nonterm, Input) :-

      Goal =.. [Nonterm, Input, []],
      call(Goal).

Kopplung mit Prolog

Allgemeines:

- Die Nichtterminalsymbol-Prädikate dürfen auch Argumente haben.

  Dies entspricht in etwa den Attributen von attributierten Grammatiken. Bei der Übersetzung werden die beiden zusätzlichen Argumente hinten angehängt.

- Man kann auf der rechten Seite normale Prolog-Aufrufe in `{...}` angeben.

  Cut, oder (das auch als `|` geschrieben werden kann) und `->` brauchen nicht in geschweifte Klammern eingeschlossen zu werden.

Beispiel:

- Die Grammatik läßt z.B. „das Lampe" zu.
- objekt `-->` artikel(G1),nomen(G2),`{`t(G1,G2)`}`.
  artikel(s) `-->` [das].
  nomen(w) `-->` [lampe].
- t(G,G) :- !.
  t(_,_) :- write('Falscher Artikel!'), fail.
- Übersetzungs-Ergebnis z.B.:
  nomen(w, S0, S1) :- S0 = [lampe|S1].

Syntax (formal)

Grammatik für Grammatiken:

- g_rule `-->` g_head, ['`-->`'], g_alternatives.
- g_head `-->` non_terminal (['`,`'], terminal | []).

  Das optionale „Kontext"-Terminalzeichen wird nach dem Parsen der rechten Seite in den Eingabestrom eingefügt. Damit kann z.B. ein „Lookahead" realisiert werden. So wird etwa „p, [a] `-->` q, [a]" übersetzt in: „p(S0, S3) :- q(S0, S1), S1 = [a|S2], S3 = [a|S2]."
- g_alternatives `-->` g_if (['`|`'], g_alternatives | []).
- g_if `-->` g_rhs, (['`->`'], g_rhs | []).
- g_rhs `-->` g_item, (['`,`'], g_rhs | []).
- g_item `-->` terminal.

  g_item `-->` non_terminal.

  g_item `-->` variable.

  g_item `-->` ['`!`'].

  g_item `-->` ['`(`'], g_alternatives, ['`)`'].

  g_item `-->` ['`(`'], prolog_goal, ['`)`'].
- non_terminal `-->` any_callable_prolog_term.
- terminal `-->` ['`[`'], (toks | []), ['`]`'].
- toks `-->` any_prolog_term, (['`,`'], toks | []).

Programmier-Stil

Effizienz-Verbesserungen:

- Einfügen von Cuts (z.B. vor Endrekursionen).
- Vermeidung von Linksrekursionen.

  Dies ist meist nicht nur ineffizient, sondern falsch: Mindestens bei inkorrekten Eingaben gerät es leicht in Endlosschleifen.
- Index über erstem Argument ausnutzen.

  ```
  lookahead(Token), [Token] --> [Token].
  stmt --> lookahead(Token), stmt(Token).
  stmt(if) --> [if], cond, [then], stmt.
  ```

Weitere Bemerkungen:

- `digit --> [C], {C >= 48, C <= 57}.`
- Besser allgemeine Grammatik-Regeln verwenden und spezifische Information in Tabellen ablegen.

  Z.B. benötigen Adventurespiel-Kommandos wie „gib“ zwei Argumente. Es ist unzweckmäßig, für jedes Verb eine eigene Grammatikregel einzuführen. Diese Information gehört eher ins Wörterbuch.
- Beispiel zur Syntaxfehler-Behandlung:

  ```
  stmt --> [T], write('Falsch:'), write(T), skip.
  skip, [';'] --> [';'], !.
  skip --> [_], !, skip.
  skip --> [].
  ```