Logisch programmeren 2011
Week 6

Michael Moortgat

17 maart 2011

Inhoudsopgave

Hoorcollege 17 maart

- Contextvrije grammatica's en DCG's
- Betekenisrepresentaties bouwen bij een DCG afleiding
- Actie op afstand: *gaps* and *fillers*

Leesmateriaal

Pereira en Shieber *Prolog and Natural Language Analysis*, Hfdst 4

Digitale editie: `http://www.mtome.com/Publications/PNLA/pnla.html`

1. CFG: contextvrije grammatica

Een contextvrije grammatica G is een 4-tal (V, Σ, R, S), met

alfabet V

$\Sigma \subseteq V$, de set van eindsymbolen; $V - \Sigma$: hulpsymbolen

R, de set van regels, is een eindige deelverzameling van $(V - \Sigma) \times V^\star$

S, het startsymbool, een element uit $V - \Sigma$

We schrijven $A \longrightarrow u$ als $(A, u) \in R$ (A een hulpsymbool, u een rijtje van hulp- en/of eindsymbolen)

$u \Rightarrow v$ is 1-stap herschrijven; $u \stackrel{\star}{\Rightarrow} v$ herschrijven in 0 of meer stappen

$L(G)$, de taal voortgebracht door G, is

$$\{w \in \Sigma^\star \mid S \stackrel{\star}{\Rightarrow} w\}$$

Voorbeeld

$$
\begin{aligned}
\text{S} &\rightarrow \text{NP VP} \\
\text{NP} &\rightarrow \text{Det N OptRel} \\
\text{NP} &\rightarrow \text{PN} \\
\text{OptRel} &\rightarrow \epsilon \\
\text{OptRel} &\rightarrow \textit{that}\ \text{VP} \\
\text{VP} &\rightarrow \text{TV NP} \\
\text{VP} &\rightarrow \text{IV}
\end{aligned}
$$

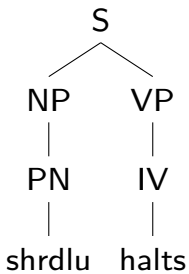
$$
\begin{aligned}
\text{PN} &\rightarrow \textit{terry} \\
\text{PN} &\rightarrow \textit{shrdlu} \\
\text{Det} &\rightarrow \textit{a} \\
\text{N} &\rightarrow \textit{program} \\
\text{IV} &\rightarrow \textit{halts} \\
\text{TV} &\rightarrow \textit{writes}
\end{aligned}
$$

Parseerbomen

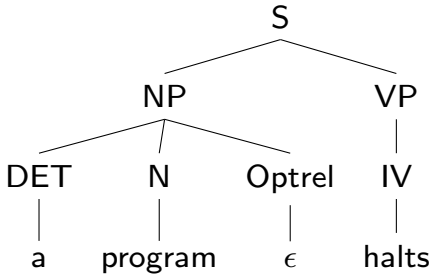
Een parseerboom is een handige grafische weergave van een afleiding $S \stackrel{\star}{\Rightarrow} w$: je abstraheert over de volgorde waarin hulpsymbolen worden herschreven.

Test hoeveel manieren heb je om shrdlu halts af te leiden?

shrdlu halts



a program halts



2. Van CFG naar Prolog

We kunnen de symbolen van een CFG op twee manieren interpreteren

als verzamelingen van uitdrukkingen (zin, naamwoordgroep, etc)

als binaire relaties over string posities

Beschouw een rijtje met posities:

$$_0\mathsf{a}_1\mathsf{program}_2\mathsf{halts}_3$$

Een regel $S \longrightarrow NP\ VP$ wordt de volgende FOL formule:

$$\forall p, p_0, p_1\ (\mathit{NP}(p_0, p_1) \wedge \mathit{VP}(p_1, p) \Rightarrow \mathit{S}(p_0, p))$$

We bewijzen die formule uit axioma's

$$\mathit{a}(0,1) \quad \mathit{program}(1,2) \quad \mathit{halts}(2,3)$$

Voorbeeld: regels

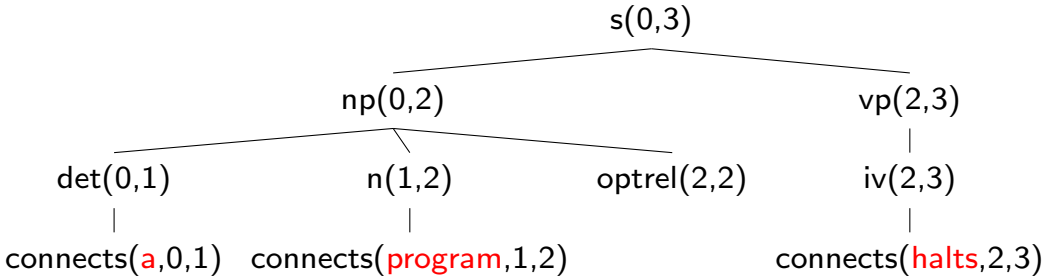
```
s(P0, P) :-
     np(P0, P1),
     vp(P1, P).
np(P0, P) :-
     det(P0, P1),
     n(P1, P2),
     optrel(P2, P).
vp(P0, P) :-
     tv(P0, P1),
     np(P1, P).
vp(P0, P) :-
     iv(P0, P).
optrel(P, P).
optrel(P0, P) :-
     connects(that, P0, P1),
     vp(P1, P).
```

Voorbeeld: eindsymbolen

Elk eindsymbool wordt ingevoerd door een literal dat een predikaat connect toepast op drie argument posities: het eindsymbool (een prolog constante) en twee argumenten die de posities aangeven

```
pn(P0, P) :-
     connects(terry, P0, P).
pn(P0, P) :-
     connects(shrdlu, P0, P).
iv(P0, P) :-
     connects(halts, P0, P).
det(P0, P) :-
     connects(a, P0, P).
n(P0, P) :-
     connects(program, P0, P).
tv(P0, P) :-
     connects(writes, P0, P).
```

Prolog als parser: query ?-s(0,3)



N.b. I.p.v. te redeneren over posities kunnen we ook redeneren over lijsten. (*op het bord*)

DCG notatie

```
s --> np,vp.
np --> det, n, optrel.
np --> pn.
vp --> tv, np.
vp --> iv.

optrel --> [].
optrel --> [that], vp.

pn --> [terry].
pn --> [shrdlu].
iv --> [halts].
det --> [a].
n --> [program].
n --> [professor].
tv --> [writes].
```

3. Extra DCG features

In een CFG zijn de hulpsymbolen atomair. In een DCG kan je extra argumenten toevoegen.

Bijvoorbeeld: Toevoegen van 'getal'-agreement

```
s --> np(Number),vp(Number).
np(Number) --> det(Number), n(Number), optrel.
np(Number) --> pn.
vp(Number) --> tv(Number), np(_).
vp(Number) --> iv(Number).

pn(sg) --> [terry].
pn(sg) --> [shrdlu].
pn(pl) --> [they].
iv(sg) --> [halts].
iv(pl) --> [halt].
tv(sg) --> [writes].
```

Extra argumenten: afleidingsboom

```
s(s(NP,VP)) --> np(NP), vp(VP).
np(np(Det,N,Rel)) --> det(Det), n(N), optrel(Rel).
np(np(PN)) --> pn(PN).
vp(vp(TV,NP)) --> tv(TV), np(NP).
vp(vp(IV)) --> iv(IV).
optrel(rel(epsilon)) --> [].
optrel(rel(that,VP)) --> [that], vp(VP).

pn(pn(terry)) --> [terry].
pn(pn(shrdlu)) --> [shrdlu].
iv(iv(halts)) --> [halts].
det(det(a)) --> [a].
n(n(program)) --> [program].
tv(tv(writes)) --> [writes].
```

Gebruik van extra argumenten (vervolg)

Boom representatie

```
?- s(Tree, [a,program,halts], []).
```

Tree = s(np(det(a), n(program), rel(epsilon)),iv(halts))

```
Tree= s(
         np(
            det(a),
            n(program),
            rel(epsilon)
            ),
          iv(halts)
          )
```

Toevoegen Prolog calls in DCG's

Versimpelen van het lexicon

Oud:

```
...
n --> [problem].
n --> [professor].
n --> [program].
...
```

Nieuw:

```
...
n --> [N], {lex_n(N)}.
lex_n(professor).
lex_n(program).
lex_n(programmer).
...
```

Toevoegen Prolog calls in DCG's (vervolg)

Toevoegen van getal-agreement

```
...
n(Number) --> [N], {lex_n(N,Number)}.

lex_n(professor,sg).
lex_n(professors,pl).
...
```

Nog korter...

```
...
n(sg) --> [N], {lex_n(N,_)}.
n(pl) --> [N], {lex_n(_,N)}.

lex_n(professor,professors).
lex_n(program,programs).
...
```

4. Semantiek voor DCG afleidingen

Parsing is niet een doel op zich. Wat we willen is

- een procedure om het ontleden van een zin te verbinden met de opbouw van een semantische representatie;
- die semantische representatie dan gebruiken voor redeneertaken

Voorbeelden van wedstrijden waarbij redeneertaken een centrale rol spelen: TREC, CLEF, RTE, ...

Compositionaliteit

Een sleutelbegrip is het beginsel van de compositionaliteit

> De betekenis van een frase is een functie van de betekenis van de delen en de manier waarop die zijn samengevoegd.

In termen van Prolog DCG regels:

```
s(SemS) --> np(SemNP), vp(SemVP),
            { assembleer(SemS,SemNP,SemVP) }.
```

waarbij `assembleer/3` aangeeft hoe je de S betekenis op kan bouwen uit de *NP* betekenis en de *VP* betekenis.

Lambda calculus

Om het proces van betekenisassemblage in volle algemeenheid uit te werken kunnen we gebruik maken van de getypeerde lambda calculus.

- De λ calculus ligt aan de basis van functionele programmeertalen zoals Haskell, ML, OCaml.
- Je zal nog uitgebreid kennis maken met de λ calculus in colleges zoals Natuurlijke Taalverwerking, Semantiek.
- In dit college beperken we ons tot een simpele benadering, waarbij we de elementaire rekenregel van de λ calculus, β reductie, simuleren met Prolog unificatie.

Abstractie, applicatie, reductie

λ abstractie is een constructie om functies aan te duiden:

$$\lambda x_1 \ldots \lambda x_n.\phi$$

staat voor een functie met parameters $x_1, \ldots x_n$; die functie geeft je de waarde van ϕ gegeven waarden voor de parameters. Voorbeeld: de `square` functie: $\lambda x.(x * x)$

Een functie kan je toepassen (applicatie) op de gewenste argumenten. Voorbeeld: $((\lambda x.(x * x))\ 3)$.

β reductie laat je toe een uitdrukking $((\lambda x.M)\ N)$ te vereenvoudigen tot $M[x \leftarrow N]$ (lees: M met alle voorkomens van x vervangen door N).

In het voorbeeld: $((\lambda x.(x * x))\ 3) \longrightarrow_\beta 3 * 3$.

5. DCG met betekenisassemblage

We gebruiken de volgende Prolog notatie voor $\lambda x_1 \ldots \lambda x_n.\phi$

```
X1^...^Xn^Phi
```

We simuleren vervolgens β reductie met unificatie: het predicaat `reduce/3` laat zich definiëren met het volgende feit:

```
% reduce/3: reduce(Functie, Argument, Resultaat)

reduce(Argument^Body, Argument, Body).
```

$\rightsquigarrow$ een nieuwe toepassing van de techniek van incomplete datastructuren (cf verschillijsten, dictionary trees, ...)

Regels met semantiek

Een paar simpele regels:

```
s(S) --> np(NP), vp(VP), {reduce(VP,NP,S)}.

vp(VP) --> tv(TV), np(NP), {reduce(TV,NP,VP)}.

tv(X^Y^wrote(Y,X)) --> [wrote].

np(shrdlu) --> [shrdlu].

np(terry) --> [terry].
```

Partiële executie

We werken de aanroep van `reduce/3` weg via pattern matching:

```
reduce(Argument^Body, Argument, Body).
```

```
s(S) --> np(NP), vp(VP), {reduce(VP,NP,S)}.
vp(VP) --> tv(TV), np(NP), {reduce(TV,NP,VP)}.
```

wordt

```
s(S) --> np(NP), vp(NP^S).
vp(VP) --> tv(NP^VP), np(NP).
```

Vergelijk: het wegwerken van `append/3` in DCG regels.

Deze techniek heet partiële executie: we verplaatsen een berekening van *run time* naar *compile time*.

Kwantificerende NPs

Onze aanpak is te simplistisch om zinnen van het type 'every program halts' aan te kunnen. In FOL, $\forall x(\mathit{program}(x) \Rightarrow \mathit{halts}(x))$. In Prolog notatie

```
all(X,program(X) => halts(X))
```

Hoe kunnen we aan de *NP* 'every program' een betekenisrepresentatie toekennen die zich compositioneel met de betekenis van de *VP* laat verbinden?

De backward engineering oplossing van Richard Montague:

every program halts	$\forall x(\mathit{program}(x) \Rightarrow \mathit{halts}(x))$
every program	$\lambda Q.\forall x(\mathit{program}(x) \Rightarrow Q(x))$
every	$\lambda P \lambda Q.\forall x(P(x) \Rightarrow Q(x))$

Montague's oplossing in Prolog

We herhalen de definitie van `reduce/3`:

```
reduce(Argument^Body, Argument, Body).
```

Stap 1 Voor het onderwerp 'every program' willen we de representatie

```
Q^all(X,program(X) ==> VP)
```

waarbij `reduce(Q,X,VP)` geldt. Partiële executie geeft

```
(X^VP)^all(X,program(X) ==> VP)
```

Stap 2 Voor het lidwoord 'every' abstraheren we op dezelfde manier over het zelfstandig naamwoord:

```
P^(X^VP)^all(X,N ==> VP)
```

waarbij `reduce(P,X,N)` geldt. Partiële executie geeft

```
(X^N)^(X^VP)^all(X,N ==> VP)
```

Montague's oplossing: DCG regels

De DCG regels voor de *NP* kunnen er nu zo uitzien:

```
np(NP) --> det(N^NP), n(N).

n(X^program(X)) --> [program].

det( (X^N)^(X^VP)^all(X,N ==> VP) ) --> [every].
```

Een voorbeeld query:

```
?-np(Sem,[every,program],[]).

Sem = (X^VP)^all(X,program(X) ==> VP)
```

DCG regels (vervolg)

In de regels voor *S* en *VP* brengen we de gewenste verhouding tussen functie en argument aan:

```
s(S) --> np(VP^S), vp(VP).

vp(X^S) --> tv(X^VP), np(VP^S).

tv(X^Y^wrote(X,Y)) --> [wrote].
```

De regel voor vp is de partiële executie van

```
vp(X^S) --> tv(TV), np(NP),
            { reduce(TV,X,VP), reduce(NP,VP,S) }.
```

Tenslotte willen we dat ook een eigennaam als functie op kan treden:

```
np( (shrdlu^S)^S ) --> [shrdlu].
```

6. Verder verkennen: bereik

Het fragment 4.2 verbindt elke zin met één enkele interpretatie.

Voor zinnen met meerdere kwantificerende elementen is dat niet genoeg.

Every student made an error

Die zin heeft twee interpretaties ('lezingen'):

```
all(X, student(X) =>
       exists(Y, error(Y) & made(X,Y))) ;
exists(Y, error(Y) &
       all(X, student(X) => made(X,Y)))
```

In vervolgcolleges (Natuurlijke Taalverwerking, Semantiek) komt deze bereiksambiguiteit uitgebreid aan de orde.

In programma 4.3 vind je alvast een voorproefje, waarbij kwantificerende uitdrukkingen niet onmiddellijk gebruikt hoeven te worden, maar voor later gebruik kunnen worden opgeslagen.

7. Actie op afstand: gaps en fillers

Vergelijk de volgende zinnen:

a Terry wrote a program that halts.
b Terry read every book that Bertrand wrote.
c Terry read every book that Bertrand told a student to write.

- Geval (a) wordt al herkend door fragment 4.2; that heeft hier betrekking op het onderwerp van de bijzin.
- In (b) en (c) heeft that betrekking op het lijdend voorwerp van wr(o|i)te.
- Geval (c) laat zien dat de afstand tussen that en het ontbrekende lijdend voorwerp opgerekt kan worden.

We willen een techniek om die afhankelijkheid over onbeperkte afstand door te geven.

Fillers and gaps

Stap 1 Gap introductie. Extra argument voor gap info.

```
np( gap(np) ) --> [].
```

Stap 2 Gaps doorgeven.

```
vp(GapInfo) --> tv, np(GapInfo).
s(GapInfo) --> np(nogap), vp(GapInfo).
```

Stap 3 Gap eliminatie. `relpron` als filler voor een frase met `gap(np)` info.

```
rel --> relpron, vp(nogap).  % program that halts
rel --> relpron, s(gap(np)). % program that terry wrote
```

Semantiek voor filler/gap afhankelijkheden

Behalve het argument voor de gap info geven we de DCG regels nu ook een argument mee voor de semantische assemblage.

Gap intro

```
np( (X^S)^S, gap(np,X) ) --> [].
```

Gap eliminatie

```
optrel( (X^S1)^X^(S1 & S2) )
   --> relpron, vp(X^S2, nogap).  % program that halts
optrel( (X^S1)^X^(S1 & S2) )
   --> relpron, s(S2, gap(np,X)). % program that terry wrote
```

In Programma 4.5 zie je het resultaat, en de behandeling van 'who/what' vragen, die hetzelfde patroon volgt.