How to write a Trale grammar

This is meant as introduction and for basic reference. See § 1 below for pointers to more detailed reading and sample grammars to get a better understanding of Trale.
Trale is an extension of ALE, an older grammar writing software, based on the programming language Prolog. Previous exposure to Prolog is a plus. Slightly modified Prolog code may be used within Trale.

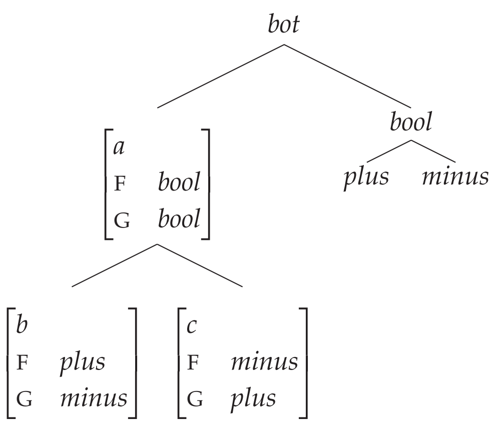
1 Links

If you haven't installed and tried Trale yet, see *How to install and run Trale*.
To learn more about Trale and Trale grammars you should consult the Trale manual.[1] and/or see the Trale download web page with more links, including more detailed documentation.[2] There is also a textbook that provides a good introduction to grammar writing in Trale.[3]

2 Signature

Signature specifies a hierarchy of types, features (attributes) appropriate to individual types, and values of these features.

(1)



The signature in (1) is written in Trale like this:

(2)

```
type_hierarchy
bot
  a f:bool g:bool
    b f:plus g:minus
    c f:minus g:plus
bool
  plus
  minus
.
```

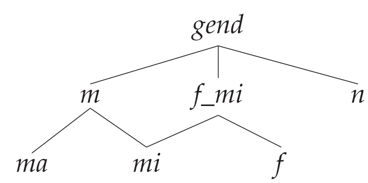
- Signature is a separate file.
- First line: `type_hierarchy`
- Last line: full stop
- Types are Prolog terms, they should start with a lower-case letter and consist of letters and numbers only.
- The most general type must be named *bot* and it is written with no indentation following the first line.
- Subtypes are introduced in separate lines, with a consistent level of increased indentation.
- Indent by two or more spaces, not by tabs.
- A type can have features, written after the type that introduces them.
- Each feature is followed by a colon and the type of its value.
- The type of the value of a feature can be specified as an arbitrary atom `a_`. This type does not need to be defined elsewhere and may be followed by `_` as the anonymous variable (3). By default, items in the PHON list are assumed to be arbitrary atoms.

(3)

```
type_hierarchy
bot
  word cat:cat phon:(a_ _)
  cat
    noun
    verb
.
```

A type can have more than one supertypes. This is an example of multiple inheritance (4).

(4)



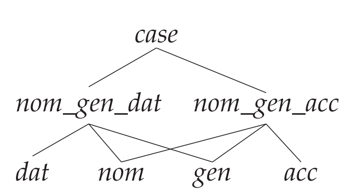
Multiple inheritance is encoded in Trale as below (5): the type occurs in the hierarchy under multiple supertypes and may be preceded for clarity by the ampersand character `&`.

(5)

```
gend
  m
    ma
    &mi
  f_mi
    f
    &mi
  n
```

Every pair of types which have a common subtype must have a unique most general common subtype. In (6) and its corresponding Trale code (7) the two types `nom_gen_dat` and `nom_gen_acc` have two common subtypes `nom` and `gen` without a unique most general subtype. The compiler reports an error (8).

(6)



[1] `http://utkl.ff.cuni.cz/~rosen/public/trale-manual.pdf`
[2] `http://milca.sfs.uni-tuebingen.de/A4/Course/trale/`
[3] `http://milca.sfs.uni-tuebingen.de/A4/Course/PDF/gramandpars.pdf`

(7)

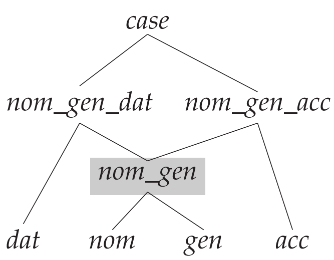
```
case
  nom_gen_dat
    dat
    &nom
    &gen
  nom_gen_acc
    &nom
    &gen
    acc
```

(8)

```
ALE: ERROR: consistent nom_gen_dat and
nom_gen_acc have multiple mgus: [nom,gen]
```

The cure is to insert the missing unique most general subtype, see (9) and the corresponding Trale code (10).

(9)



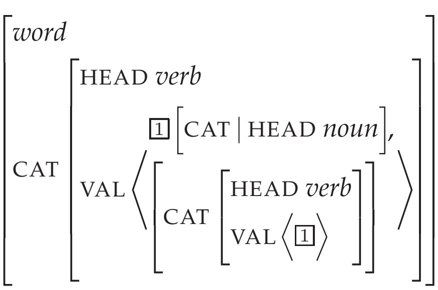
(10)

```
case
  nom_gen_dat
    dat
    &nom_gen
      nom
      gen
  nom_gen_acc
    &nom_gen
    acc
```

3 Theory

Theory is the name for the 'real' Trale grammar, using the concepts specified in *signature*. It is a separate file, named `theory`. Signature provides possible objects used to model language entities. Theory restricts these objects by lexical entries, phrase structure rules, principles, relational constraints, lexical rules and a few other constructs. To pick the right set of objects, the constructs use *descriptions*. Complex descriptions can be formed using logical connectives, the comma for logical conjunction being the most frequent connective.
The graphical description in (11) is expressed in Trale as (12).

(11)



(12)

```
(word,
 cat:(head:verb,
      val:[(X,cat:head:noun),
           (cat:(head:verb,
                 val:[X]))]))
```

Comma within a list is not a logical connective but separates list items. The boxed number preceding a feature structure is expressed as a conjuction of the variable and the feature structure.

3.1 Descriptions

`<desc> ::=`
`<type>`
... according to signature, refers to all objects of that type
`| <variable>`
... starts with upper case, refers to any object, multiple occurrences of the same variable must refer to the same variable
`| (<feature>:<desc>)`
... picks out objects whose value for the feature satisfies the nested description
`| (<desc>,<desc>)`
... logical conjuction
`| (<desc>;<desc>)`
... logical disjunction
`| @ <macro_spec>`
... macro call
`| <func_spec>`
... function
`| a_ <prolog_term>`
... atom not defined in signature
`| <path> == <path>`
... equation, satisfied by objects that are token-identical
`| (=\= <desc>)`
... inequation, satisfied by objects that are not token-identical to objects described by <desc>, a space should occur between =\= and a preceding operator
`([arg1]==[arg2])` is equivalent to `(arg1:X,arg2:X)`.

Operator precedence and association

```
a, b ; c, d ; e = (a,b);(c,d);e
a,b,c = a,(b,c)
f:g:bot,h:j = (f:(g:bot)),(h:j)
f:g: =\=k,h:j = (f:(g: =\=(k))),(h:j)
f:[g]==[h],h:j = (f:([g]==[h])),(h:j)
```

3.2 Lexical entries

`<atom> ~~> <desc>`
The `<atom>` is the orthography of the word, by default it becomes the single item of the PHON list.

```
kluk ~~> (word,
          cat:(noun,
               pdgm:(a_ pán),
               case:nom,
               agr:(num:sg,
                    gend:ma))).
```

3.3 Phrase structure rules

The rules are constraints of a special type: they implicitly relate the mother's and the daughter's PHON lists. They are equivalent to rules of a context-free phrase structure grammar: the mother's PHON list is the concatenation of the daughter's PHON lists, in the order of their appearance in the rule.

3.3.1 Simple rules

```
<rule> ::=
  <rule_name> ##
  <desc> ===>
  cat> <desc>
  cat> <desc>.
```

The `<desc>` following ## stands for the syntactic mother. The `<desc>` following `cat>` stands for a syntactic daughter.

```
headfin_c ##
    (hc_phrase,
     h_init:minus,
     head_dtr:Head,
     nonh_dtr:NonH)
===>
    cat> NonH,
    cat> Head.
```

The rule above can be depicted like this:

$$\begin{bmatrix} \textit{hc\_phrase} & \\ \text{H\_INIT} & \textit{minus} \\ \text{HEAD\_DTR} & \boxed{1} \\ \text{NONH\_DTR} & \boxed{2} \end{bmatrix} \longrightarrow \boxed{1} \;\; \boxed{2}$$

3.3.2 More complex rules

```
<rule> ::=
  <rule_name> ## <desc> ===> <rule_body>.
<rule_body> ::=
  <rule_clause>
  | <rule_clause>, <rule_body>
<rule_clause> ::=
  cat> <desc>
  | cats> <desc>
  | goal> <goal>
```

The `<desc>` following `cats>` gives a list of syntactic daughters. The `<goal>` stands for a procedural attachment.

```
schema2 ##
(cat:(head:Head,
      subcat:[Subj]))
===>
cat>
(cat:(head:Head,
      subcat:[Subj|Comps])),
cats> Comps.

backward_application rule
(synsem:Z,
 qstore:Qs)
===>
cat>
(synsem:Y,
 qstore:Qs1),
cat>
(synsem:(backward,
 arg:Y,
 res:Z),
 qstore:Qs2),
goal>
append(Qs1,Qs2,Qs).
```

3.4 Principles

Principles are equivalent to logical implications, they apply to all objects, provided that the object matches the antecedent.

```
<principle> ::=
  <desc> *> <princ_clause>.
<princ_clause> ::=
  <desc>
  | <desc> <goal>
```

The `<desc>` before `*>` stands for a description without functions or inequations.

```
(val,subj:ne_list) *> subj:[_].
```

3.5 Macros

Macros are used to abbreviate repeatedly occuring descriptions. They can have any number of arguments, and the arguments can be "guarded" by a type – the argument must then be of that type.

```
  n(X-case) := (word, cat:(noun, case:X)).
  pepa ~~> @n(nom).
```

The macro is defined as having one argument, and the argument must be of the type *case*. The macro is called in a lexical entry.

3.6 Relational constraints

Relational constraints give Trale the power of the programming language Prolog. They can be defined in a usual way and invoked in a `goal` clause, or in a functional notation and used inside descriptions at the position where the result should occur.

3.6.1 Relations

```
append([],L,L) if true.
append([H|T],L,[H|Res]) if append(T,L,Res).
```

Some relations can take a long time to evaluate or they may prevent the parsing process from terminating. A prudent grammar writer includes `when/2` clauses to delay the evaluation of a relation before enough information is known about its arguments.

```
append(X,Y,Z) if
   when(  ( X=(e_list;ne_list)
          ; Y=e_list
          ; Z=(e_list;ne_list)
              ),
            undelayed_append(X,Y,Z)).

undelayed_append(L,[],L) if true.
undelayed_append([],(L,ne_list),L) if true.
undelayed_append([H|T1],(L,ne_list),[H|T2]) if
  append(T1,L,T2).
```

3.6.2 Functional notation

```
% append(+,+,-) This append assumes that the first or the third
% argument are known to be non_empty or empty lists.
%

fun append(+,+,-).
append(X,Y,Z) if
   when( (X=(e_list;ne_list);
          Z=(e_list;ne_list))
       , undelayed_append(X,Y,Z)
       ).

undelayed_append([],L,L) if true.
undelayed_append([H|T1],L,[H|T2]) if append(T1,L,T2).
```

3.7 Lexical Rules

Lexical rules are used to derive lexical entries at compile time from those already specified. They can be used to derive word forms from a base form or other word forms. The description on the left-hand side is replaced by the description on the left-hand side, any identities must be explicitly mentioned.

```
<lex_rule> ::=
  <lex_rule_name> lex_rule <lex_rewrite>
  morphs <morphs>.
<lex_rewrite> ::=
  <desc> **> <desc>
  | <desc> **> <desc> if <goal>
<morphs> ::=
  <morph>
  | <morph>, <morphs>
<morph> ::=
  (<string_pattern>) becomes (<string_pattern>)
  | (<string_pattern>) becomes (<string_pattern>)
    when <prolog_goal>
<string_pattern> ::=
  <atomic_string_pattern>
  | <atomic_string_pattern>, <string_pattern>
<atomic_string_pattern> ::=
  <atom>
  | <var>
  | <list(<var_char>)>
<var_char> ::= <char>
```

The following example rule derives English plural nouns and includes the definition of a relational constraint.

```
plural_n lex_rule
(n,
 num:sing)
**>
(n,
 num:plu)
morphs
goose becomes geese,
[k,e,y] becomes [k,e,y,s],
(X,man) becomes (X,men),
(X,F) becomes (X,F,es) when fricative(F),
(X,ey) becomes (X,[i,e,s]),
X becomes (X,s) if true.

fricative([s]) if true.
fricative([c,h]) if true.
fricative([s,h]) if true.
fricative([x]) if true.
```

The `becomes` clauses can also be replaced by a single clause invoking an appropriate relational constraint:
`X becomes Y when morph_plural(X,Y).`

3.8 Comments

Comments are preceded by `%`

3.9 Display options

3.9.1 Hiding features

Features that should not be displayed in the graphical interface.

```
hidden_feat(dtrs).
```

3.9.2 Feature ordering

Alters the default alphabetic ordering in the graphical interface.
`f <<< g.` Meaning: f will be ordered before g.
`<<< h.` Meaning: h will be ordered last.
`>>> i.` Meaning: i will be ordered first.

3.10 Test sequences

Test items are encoded as `t/5` facts:

```
t(Nr,‘‘Test Item’’,Desc,ExpSols,’Comment’).
```

`Nr`: test item ID number
`Test Item`: test string, must be enclosed in double-quotes
`Desc`: optional start category description, leave uninstantiated to get all possible parses
`Comment`: optional comment, enclosed in single-quotes

3.11 Technical specifications

may be version-dependent

```
:- discontiguous ’*>’/2.
:- discontiguous ’fun’/1.
:- discontiguous ’if’/2.
:- tree_extensions.
:- multifile if/2.
```