GF Quick Reference

Aarne Ranta, April 4, 2006

This is a quick reference on GF grammars. It aims to cover all forms of expression available when writing grammars. It assumes basic knowledge of GF, which can be acquired from the *GF Tutorial* (`http://www.cs.chalmers.se/~aarne/GF/doc/tutorial/`). Help on GF commands is obtained on line by the help command (`help`), and help on invoking GF with (`gf -help`).

A Complete Example

This is a complete example of a GF grammar divided into three modules in files. The grammar recognizes the phrases *one pizza* and *two pizzas.*

File `Order.gf`:

```
abstract Order = {
cat
  Order ;
  Item ;
fun
  One, Two : Item -> Order ;
  Pizza : Item ;
}
```

File `OrderEng.gf` (the top file):

```
--# -path=.:prelude
concrete OrderEng of Order =
 open Res, Prelude in {
flags startcat=Order ;
lincat
  Order = SS ;
  Item  = {s : Num => Str} ;
lin
  One it = ss ("one" ++ it.s ! Sg) ;
  Two it = ss ("two" ++ it.s ! Pl) ;
  Pizza  = regNoun "pizza" ;
}
```

File `Res.gf`:

```
resource Res = open Prelude in {
param Num = Sg | Pl ;
oper regNoun : Str -> {s : Num => Str} =
  \dog -> {s = table {
    Sg => dog ;
    _  => dog + "s"
    }
  } ;
}
```

To use this example, do

```
  % gf             -- in shell: start GF
  > i OrderEng.gf  -- in GF: import grammar
  > p "one pizza"  --        parse string
  > l Two Pizza    --        linearize tree
```

Modules and files

One module per file. File named `Foo.gf` contains module named `Foo`.

Each module has the structure

```
moduletypename =
  Inherits **         -- optional
  open Opens in       -- optional
  { Judgements }
```

Inherits are names of modules of the same type. Inheritance can be restricted:

```
  Mo[f,g],  -- inherit only f,g from Mo
  Lo-[f,g]  -- inheris all but f,g from Lo
```

Opens are possible in `concrete` and `resource`. They are names of modules of these two types, possibly qualified:

```
  (M = Mo), -- refer to f as M.f or Mo.f
  (Lo = Lo) -- refer to f as Lo.f
```

Module types and judgements in them:

```
abstract A          -- cat, fun, def, data
concrete C of A     -- lincat, lin, lindef, printname
resource R          -- param, oper

interface I         -- like resource, but can have
                       oper f : T without definition
instance J of I     -- like resource, defines opers
                       that I leaves undefined
incomplete          -- functor: concrete that opens
 concrete CI of A =    one or more interfaces
  open I in ...
concrete CJ of A =  -- completion: concrete that
  CI with              instantiates a functor by
    (I = J)            instances of open interfaces
```

The forms `param`, `oper` may appear in `concrete` as well, but are then not inherited to extensions.

All modules can moreover have `flags` and comments. Comments have the forms

```
-- till the end of line
{- any number of lines between -}
--# used for compiler pragmas
```

A concrete can be opened like a resource. It is translated as follows:

```
cat C           --->  oper C : Type =
lincat C = T            T ** {lock_C : {}}

fun f : G -> C --->  oper f : A* -> C* = \g ->
lin f = t               t g ** {lock_C = <>}
```

An abstract can be opened like an interface. Any concrete of it then works as an instance.

Judgements

```
cat C                 -- declare category C
cat C (x:A)(y:B x)  -- dependent category C
cat C A B             -- same as C (x : A)(y : B)
fun f : T             -- declare function f of type T
def f = t             -- define f as t
def f p q = t         -- define f by pattern matching
data C = f | g        -- set f,g as constructors of C
data f : A -> C       -- same as
                         fun f : A -> C; data C=f

lincat C = T          -- define lin.type of cat C
lin f = t             -- define lin. of fun f
lin f x y = t         -- same as lin f = \x y -> t
lindef C = \s -> t  -- default lin. of cat C
printname fun f = s -- printname shown in menus
printname cat C = s -- printname shown in menus
printname f = s     -- same as printname fun f = s

param P = C | D Q R -- define parameter type P
                         with constructors
                         C : P, D : Q -> R -> P
oper h : T = t        -- define oper h of type T
oper h = t            -- omit type, if inferrable

flags p=v             -- set value of flag p
```

Judgements are terminated by semicolons (;). Subsequent judgments of the same form may share the keyword:

```
cat C ; D ;           -- same as cat C ; cat D ;
```

Judgements can also share RHS:

```
fun f,g : A           -- same as fun f : A ; g : A
```

Types

Abstract syntax (in fun):

```
C                  -- basic type, if cat C
C a b              -- basic type for dep. category
(x : A) -> B       -- dep. functions from A to B
(_ : A) -> B       -- nondep. functions from A to B
(p,q : A) -> B     -- same as (p : A)-> (q : A) -> B
A -> B             -- same as (_ : A) -> B
Int                -- predefined integer type
Float              -- predefined float type
String             -- predefined string type
```

Concrete syntax (in lincat):

```
Str                -- token lists
P                  -- parameter type, if param P
P => B             -- table type, if P param. type
{s : Str ; p : P}-- record type
{s,t : Str}        -- same as {s : Str ; t : Str}
{a : A} **{b : B}-- record type extension, same as
                      {a : A ; b : B}
A * B * C          -- tuple type, same as
                      {p1 : A ; p2 : B ; p3 : C}
Ints n             -- type of n first integers
```

Resource (in oper): all those of concrete, plus

```
Tok                -- tokens (subtype of Str)
A -> B             -- functions from A to B
Int                -- integers
Strs               -- list of prefixes (for pre)
PType              -- parameter type
Type               -- any type
```

As parameter types, one can use any finite type: P defined in param P, Ints n, and record types of parameter types.

Expressions

Syntax trees = full function applications

```
f a b               -- : C if fun f : A -> B -> C
1977                -- : Int
3.14                -- : Float
"foo"               -- : String
```

Higher-Order Abstract syntax (HOAS): functions as arguments:

```
F a (\x -> c)       -- : C if a : A, c : C (x : B),
                       fun F : A -> (B -> C) -> C
```

Tokens and token lists

```
"hello"             -- : Tok, singleton Str
"hello" ++ "world" -- : Str
["hello world"]     -- : Str, same as "hello" ++ "world"
"hello" + "world"   -- : Tok, computes to "helloworld"
[]                  -- : Str, empty list
```

Parameters

```
Sg                    -- atomic constructor
VPres Sg P2           -- applied constructor
{n = Sg ; p = P3}     -- record of parameters
```

Tables

```
table {               -- by full branches
  Sg => "mouse" ;
  Pl => "mice"
  }
table {               -- by pattern matching
  Pl => "mice" ;
  _  => "mouse"       -- wildcard pattern
  }
table {
  n => regn n "cat"   -- variable pattern
  }
table Num {...}       -- table given with arg. type
table ["ox"; "oxen"]  -- table as course of values
\\_ => "fish"         -- same as table {_ => "fish"}
\\p,q => t            -- same as \\p => \\q => t

t ! p                 -- select p from table t
case e of {...}       -- same as table {...} ! e
```

Records

```
{s = "Liz"; g = Fem} -- record in full form
{s,t = "et"}          -- same as {s = "et";t= "et"}
{s = "Liz"} **        -- record extension: same as
  {g = Fem}              {s = "Liz" ; g = Fem}

<a,b,c>        -- tuple, same as {p1=a;p2=b;p3=c}
```

Functions

```
\x -> t              -- lambda abstract
\x,y -> t            -- same as \x -> \y -> t
\x,_ -> t            -- binding not in t
```

Local definitions

```
let x : A = d in t -- let definition
let x = d in t     -- let defin, type inferred
let x=d ; y=e in t -- same as
                      let x=d in let y=e in t
let {...} in t     -- same as let ... in t

t where {...}      -- same as let ... in t
```

Free variation

```
variants {x ; y}      -- both x and y possible
variants {}           -- nothing possible
```

Prefix-dependent choices

```
pre {"a" ; "an" / v} -- "an" before v, "a" otherw.
strs {"a" ; "i" ;"o"}-- list of condition prefixes
```

Typed expression

```
<t:T>                 -- same as t, to help type inference
```

Accessing bound variables in `lin`: use fields `$1`, `$2`, `$3`, ... Example:

```
fun F : (A : Set) -> (El A -> Prop) -> Prop ;
lin F A B = {s = ["for all"] ++ A.s ++ B.$1 ++ B.s}
```

Pattern matching

These patterns can be used in branches of `table` and `case` expressions. Patterns are matched in the order in which they appear in the grammar.

```
C                  -- atomic param constructor
C p q              -- param constr. applied to patterns
x                  -- variable, matches anything
_                  -- wildcard, matches anything
"foo"              -- string
56                 -- integer
{s = p ; y = q}    -- record, matches extensions too
<p,q>              -- tuple, same as {p1=p ; p2=q}
p | q              -- disjunction, binds to first match
x@p                -- binds x to what p matches
- p                -- negation
p + "s"            -- sequence of two string patterns
p*                 -- repetition of a string pattern
```

Sample library functions

```
-- lib/prelude/Predef.gf
drop   : Int -> Tok -> Tok   -- drop prefix of length
take   : Int -> Tok -> Tok   -- take prefix of length
tk     : Int -> Tok -> Tok   -- drop suffix of length
dp     : Int -> Tok -> Tok   -- take suffix of length
occur  : Tok -> Tok -> PBool -- test if substring
occurs : Tok -> Tok -> PBool -- test if any char occurs
show   : (P:Type) -> P ->Tok -- param to string
read   : (P:Type) -> Tok-> P -- string to param
toStr  : (L:Type) -> L ->Str -- find "first" string

-- lib/prelude/Prelude.gf
param Bool = True | False
oper
  SS  : Type                     -- the type {s : Str}
  ss  : Str -> SS                -- construct SS
  cc2 : (_,_ : SS) -> SS         -- concat SS's
  optStr : Str -> Str            -- string or empty
  strOpt : Str -> Str            -- empty or string
```

```
bothWays : Str -> Str -> Str -- X++Y or Y++X
init : Tok -> Tok            -- all but last char
last : Tok -> Tok            -- last char
prefixSS : Str -> SS -> SS
postfixSS : Str -> SS -> SS
infixSS : Str -> SS -> SS -> SS
if_then_else : (A : Type) -> Bool -> A -> A -> A
if_then_Str : Bool -> Str -> Str -> Str
```

Flags

Flags can appear, with growing priority,

- in files, judgement `flags` and without dash (`-`)
- as flags to `gf` when invoked, with dash
- as flags to various GF commands, with dash

Some common flags used in grammars:

```
startcat=cat    use this category as default

lexer=literals  int and string literals recognized
lexer=code      like program code
lexer=text      like text: spacing, capitals
lexer=textlit   text, unknowns as string lits

unlexer=code    like program code
unlexer=codelit code, remove string lit quotes
unlexer=text    like text: punctuation, capitals
unlexer=textlit text, remove string lit quotes
unlexer=concat  remove all spaces
unlexer=bind    remove spaces around "&+"

optimize=all_subs  best for almost any concrete
optimize=values    good for lexicon concrete
optimize=all       usually good for resource
optimize=noexpand  for resource, if =all too big
```

For the full set of values for `FLAG`, use on-line `h -FLAG`.

File paths

Colon-separated lists of directories searched in the given order:

```
--# -path=.:../abstract:../common:prelude
```

This can be (in order of growing preference), as first line in the top file, as flag to `gf` when invoked, or as flag to the `i` command. The prefix `--#` is used only in files.

If the environment variabls `GF_LIB_PATH` is defined, its value is automatically prefixed to each directory to extend the original search path.

Alternative grammar formats

Old GF (before GF 2.0): all judgements in any kinds of modules, division into files uses `includes`. A file `Foo.gf` is recognized as the old format if it lacks a module header.

Context-free (file `foo.cf`). The form of rules is e.g.

```
Fun. S ::= NP "is" AP ;
```

If `Fun` is omitted, it is generated automatically. Rules must be one per line. The RHS can be empty.

Extended BNF (file `foo.ebnf`). The form of rules is e.g.

```
S ::= (NP+ ("is" | "was") AP | V NP*) ;
```

where the RHS is a regular expression of categories and quoted tokens: `"foo", CAT, T U, T|U, T*, T+, T?`, or empty. Rule labels are generated automatically.

Probabilistic grammars (not a separate format). You can set the probability of a function `f` (in its value category) by

```
--# prob f 0.009
```

These are put into a file given to GF using the `probs=File` flag on command line. This file can be the grammar file itself.

Example-based grammars (file `foo.gfe`). Expressions of the form

```
in Cat "example string"
```

are preprocessed by using a parser given by the flag

```
--# -resource=File
```

and the result is written to `foo.gf`.

References

GF Homepage (`http://www.cs.chalmers.se/~aarne/GF/`)

A. Ranta, Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming*, vol. 14:2. 2004, pp. 145-189.