Embedded Grammars

Aarne Ranta

GF Summer School, 25 August 2009

Contents

Grammars in language technology applications

A grammar is capable of certain things

- parsing, linearization, translation
- type checking, dialogue management
- graphic rendering
- speech recognition

The rest - or even some of this - is done in a **host language**

Example: a dialogue system

input speech recognition	Nuance, HTK	from GF
input parsing	PGF interpreter	in GF
dialogue management	VoiceXML, TrindiKit	maybe from GF
database access	SQL	hardly in GF
inference	theorem prover	hardly in GF
response generation	PGF interpreter	in GF
speech synthesis	Festival, Nuance	not in GF

Wanted: easy integration of GF and host language programming

The PGF format

PGF = Portable Grammar Format

A variant of PMCFG

Multilingual grammar: abstract + concretes

Low-level, binary files

Generated by `gf -make`

The API for the Haskell interpreter of PGF

```
readPGF   :: FilePath -> IO PGF

linearize :: PGF -> Language -> Tree -> String
parse     :: PGF -> Language -> Category -> String -> [Tree]

linearizeAll     :: PGF -> Tree -> [String]
linearizeAllLang :: PGF -> Tree -> [(Language,String)]

parseAll     :: PGF -> Category -> String -> [[Tree]]
parseAllLang :: PGF -> Category -> String -> [(Language,[Tree])]

languages    :: PGF -> [Language]
categories   :: PGF -> [Category]
startCat     :: PGF -> Category
```

A batch translator

Functionality: translation from English to Italian with standard input and output.

```
$ echo "this wine is delicious" | ./trans Food.pgf
questo vino è delizioso
```

Translator code in Haskell

```
module Main where

import PGF
import System (getArgs)

main :: IO ()
main = do
  file:_ <- getArgs
  gr     <- readPGF file
  interact (translate gr)

translate :: PGF -> String -> String
translate gr s = case parseAllLang gr (startCat gr) s of
  (lg,t:_):_ ->
    unlines [linearize gr l t | l <- languages gr, l /= lg]
  _ -> "NO PARSE"
```

Building the translator

Run the Haskell compiler GHC to produce the executable `trans`:

```
unix$ ghc --make -o trans Translator.hs
```

Run the GF compiler to produce the PGF file `Foods.pgf`:

```
unix$ gf -make FoodEng.gf FoodIta.gf
```

Generalized translation function

```
translate :: (Tree -> Tree) -> PGF -> String -> String
translate f pgf = linearize pgf . f . parse pgf
```

A query system

Functionality: questions and answers, in the same language.

```
unix$ ./query
is 123 prime
No.
onko 6 pariton
Ei.
quit
bye
unix$
```

Code: the main loop

```
module Main where

import PGF
import Answer (transfer)

main :: IO ()
main = do
  gr <- readPGF "Query.pgf"
  loop (translate transfer gr)

loop :: (String -> String) -> IO ()
loop trans = do
  s <- getLine
  if s == "quit" then putStrLn "bye" else do
    putStrLn $ trans s
    loop trans

translate :: (Tree -> Tree) -> PGF -> String -> String
translate tr gr s = case parseAllLang gr (startCat gr) s of
  (lg,t:_):_ -> linearize gr lg (tr t)
  _ -> "NO PARSE"
```

Code: the abstract syntax

GF module and its automatic translation to Haskell datatypes

```
abstract Query = {
  cat Answer ; Question ;
  fun Yes    : Answer ;
      No     : Answer ;

  cat Question ;
  fun Even   : Object -> Question ;
      Odd    : Object -> Question ;
      Prime  : Object -> Question ;

  cat Object ;
  fun Number : Int -> Object ;
}
```

```
data Answer =
   GYes
 | GNo

data Question =
   GEven  Object
 | GOdd   Object
 | GPrime Object

data Object =
   GNumber GInt
```

Generated by `gf -make --output-format=haskell Query.gf`

Code: mapping between GF trees and Haskell data

```
  class Gf a where
    gf :: a -> Tree
    fg :: Tree -> a

instance Gf GAnswer where
  gf GNo = Fun (mkCId "No") []
  gf GYes = Fun (mkCId "Yes") []

  fg t =
    case t of
      Fun i [] | i == mkCId "No" -> GNo
      Fun i [] | i == mkCId "Yes" -> GYes
      _ -> error ("no Answer " ++ show t)
```

Code: the answer generator

```
module Answer where

import PGF (Tree)
import Query

transfer :: Tree -> Tree
transfer = gf . answer . fg

answer :: GQuestion -> GAnswer
answer p = case p of
  GOdd x   -> test odd x
  GEven x  -> test even x
  GPrime x -> test prime x

value :: GObject -> Int
value e = case e of
  GNumber (GInt i) -> fromInteger i

test :: (Int -> Bool) -> GObject -> GAnswer
test f x = if f (value x) then GYes else GNo

prime :: Int -> Bool
prime x = elem x primes where
  primes = sieve [2 .. x]
  sieve (p:xs) = p : sieve [ n | n <- xs, n `mod` p > 0 ]
  sieve [] = []
```

Putting it all together

Files:

```
Makefile          -- a makefile
Query.gf          -- abstract syntax
Query???.gf       -- concrete syntaxes
Answer.hs         -- definition of question-to-answer function
QuerySystem.hs    -- Haskell Main module
```

To make:

```
all:
        gf -make --output-format=haskell Query???.gf
        ghc --make -o query QuerySystem.hs
```

Code generation from GF

Nuance speech recognition (GSL):

```
unix$ gf -make --output-format=gsl FoodsEng.gf
```

Finite automaton:

```
unix$ gf -make --output-format=fa FoodsDut.gf

unix$ dot -Tpng FoodsDut.dot >FoodsDut.png
```