Linearisation-only PGF

GF Summer School 2021

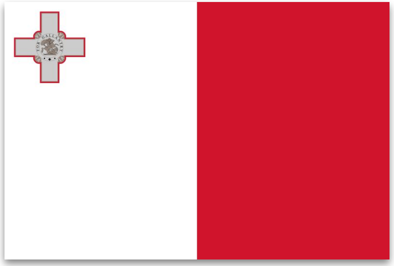
John J. Camilleri

digital Grammars

Language technology to rely on.

About me

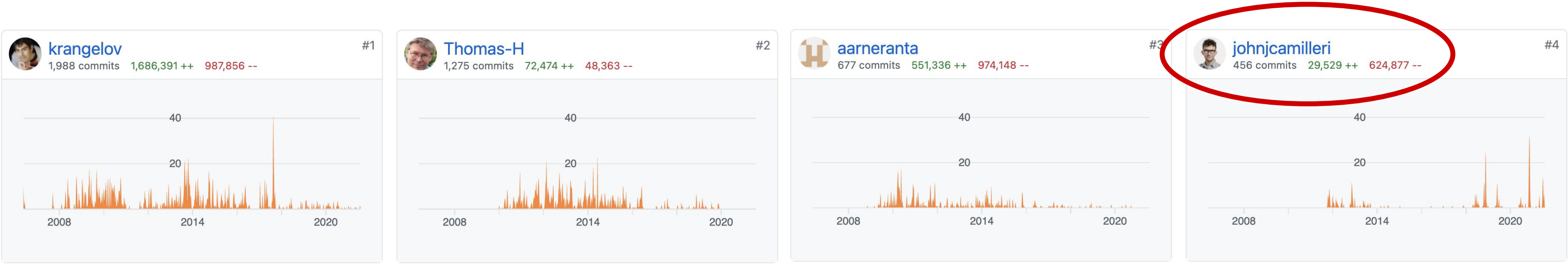
- From Malta
- Attended first GF summer school in Gothenburg, 2009
- Ph.D. in Computer Science at Chalmers & University of Gothenburg
- CTO at Digital Grammars



GF Summer School
Gothenburg, Sweden
August 17-28, 2009

UNIVERSITY OF
GOTHENBURG

Me and GF



Source: https://github.com/GrammaticalFramework/gf-core/graphs/contributors

- Worked with GF on/off for 12 years
- Some grammars, but mostly integration and internal stuff
- At DG we use GF in the "real world"
- What I've learnt is that...

parsing is hard

Problems with parsing

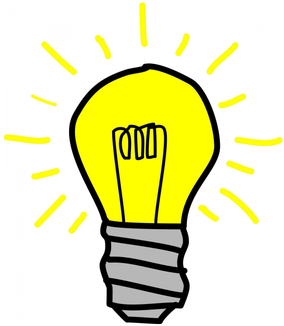
- Ambiguity is impossible to avoid
  - probabilities?
  - statistical models?
  - exceptions?
- Fitting grammar to a corpus is unsatisfying work
- Corpus is often wrong, inconsistent, or otherwise problematic
- Coverage comes at cost of overgeneration

generation is easy

What's nice about generation

- It's not parsing
- There's no ambiguity (if you ignore variants)
- Data is usually more structured & consistent
- If customer data already exists, half the work is already done
- If not, designing abstract data more *is* a satisfying task
- Much better control of the process
- Generation is a "simpler" operation

Motivation behind LPGF



1. GF grammars enable both parsing and generation via PGF.
2. Compiling to PGF can be slow & memory-intensive.
3. Often we don't want to parse at all.
4. Let's compile to something smaller which only supports linearisation: LPGF.
5. Anticipated benefits:
   a. Compiling GF to LPGF will be less resource-demanding than to PGF
   b. An LPGF file will be smaller than a PGF file
   c. The speed of linearisation will be faster and require less memory
   d. Not needing to support parsing can open up other possible features
6. The theory already exists!

Published: 15 December 2009

PGF: A Portable Run-time Format for Type-theoretical Grammars

Krasimir Angelov, Björn Bringert & Aarne Ranta

Journal of Logic, Language and Information **19**, 201–228 (2010) | Cite this article

Abstract

Portable Grammar Format (PGF) is a core language for type-theoretical grammars. It is the target language to which grammars written in the high-level formalism Grammatical

A bit of history

Paper (2009)	Today (2021)
PGF (section 2)	LPGF
PMCFG (section 3)	PGF

- Initially:
  - PGF for linearisation
  - PMCFG for parsing
  - GF compiled to both
- Later discovered that:
  - PGF can also be used for parsing (but very inefficient)
  - PMCFG can also be used for linearisation (but has size issues)
- Currently:
  - GF is compiled only to PMCFG
  - Makes compiler and runtime simpler (with some tradeoffs in performance and features)
- Proposal:
  - Revive linearisation-only PGF
  - Pure generation is a common use case
  - Reclaim performance tradeoffs & add extra features

The situation has been clear for a long time:

- *PMCFG-only is a simpler solution to implement*
- *but it is too costly for some grammars,*
  and unnecessarily so if only linearization is needed
- *some tasks that one could in principle perform with GF*
  are therefore simply not possible in practice

Aarne Ranta, personal communication (2021)

I'm sold!

down to the nitty-gritty...

LPGF: syntax

$$t ::= [\,] \mid \text{“foo”} \mid t_1 \mathbin{+\!\!+} t_2 \mid i \mid <t_1, \ldots, t_n> \mid t_1 \,!\, t_2 \mid \$_i$$

empty string

token

concatenation

numeric constant

tuple

projection

argument variable

LPGF: type system

Strings:

$$[\,] : \mathsf{Str} \quad \text{“foo”} : \mathsf{Str} \quad \frac{s, t : \mathsf{Str}}{s +\!\!+\, t : \mathsf{Str}}$$

Bounded integers:

$$i : \mathsf{Int}_i \quad \frac{i : \mathsf{Int}_m}{i : \mathsf{Int}_n}\; m < n$$

Tuples:

$$\frac{t_1 \;:\; T_1 \;\ldots\; t_n \;:\; T_n}{< t_1, \ldots, t_n >:\; T_1 * \cdots * T_n}$$

Projections:

$$\frac{t : \mathsf{T}^n \;\; u : \mathsf{Int}_n}{t \,!\, u : \mathsf{T}} \quad \frac{t : \mathsf{T}_1 * \ldots * \mathsf{T}_n}{t \,!\, i : \mathsf{T}_i}\; i = 1, \ldots, n$$

Argument variables:

$$\frac{}{T_1, \ldots, T_n \vdash \$_i : \mathsf{T}_i}\; i = 1, \ldots, n$$

LPGF: operational semantics $\Downarrow$

Strings:

$$[\,] \Downarrow [\,] \quad \text{“foo”} \Downarrow \text{“foo”} \quad \frac{s \Downarrow v \quad t \Downarrow w}{s \mathbin{+\!\!+} t \Downarrow v \mathbin{+\!\!+} w}$$

Bounded integers:

$$i \Downarrow i$$

Tuples:

$$\frac{t_1 \Downarrow v_1 \ \ldots \ t_n \Downarrow v_n}{< t_1, \ldots, t_n > \Downarrow < v_1, \ldots, v_n >}$$

Projections:

$$\frac{t \Downarrow < v_1, \ldots, v_n > \quad u \Downarrow i}{t!u \Downarrow v_i} \ i = 1, \ldots, n$$

Argument variable:

$$v_1, \ldots, v_n \vdash \$_i \Downarrow v_i \ (i = 1, \ldots, n)$$

LPGF: linearisation operation $\mapsto$

$$\frac{a_1 \mapsto t_1 \quad \ldots \quad a_n \mapsto t_n \quad t_1,\ldots,t_n \vdash t \Downarrow v}{f\,a_1 \ldots a_n \mapsto v}\,\mathbf{lin}\, f = t$$

Implementation: starting point

- LPGF datatype is implemented as:
  `LPGF` (src/runtime/haskell/LPGF.hs)

- GF parser gives you a term of type:
  `SourceGrammar` (src/compiler/GF/Grammar/Grammar.hs)

- Top-level PGF compile function has type:
  `mkCanon2pgf  :: ... SourceGrammar -> IO PGF`

- Top-level LPGF compile function has type:
  `mkCanon2lpgf :: ... SourceGrammar -> IO LPGF`

At some point we supported both formats and used them for different purposes. The problem was that the compilation to linearization-only format was full of bugs. After I spent many hours of fixing the compiler, I just dropped it and started using the parsing-only format for everything.

Krasimir Angelov, personal communication (2021)

At some point we supported both formats and used them for different purposes. The problem was that the compilation to linearization-only format was full of bugs. After I spent many hours of fixing the compiler, I just dropped it and started using the parsing-only format for everything.

I remember there were some problems, but some of them were due to the normalization of expressions not working properly. Now that Thomas [Hallgren] fixed it long ago (normalization by evaluation), the task should be easier.

GF Canonical format

src/compiler/GF/Grammar/Canonical.hs

- A subset of GF
- What's left after high-level constructions such as functors and opers have been eliminated by partial evaluation.
- Intended as a common intermediate representation to simplify export to other formats.

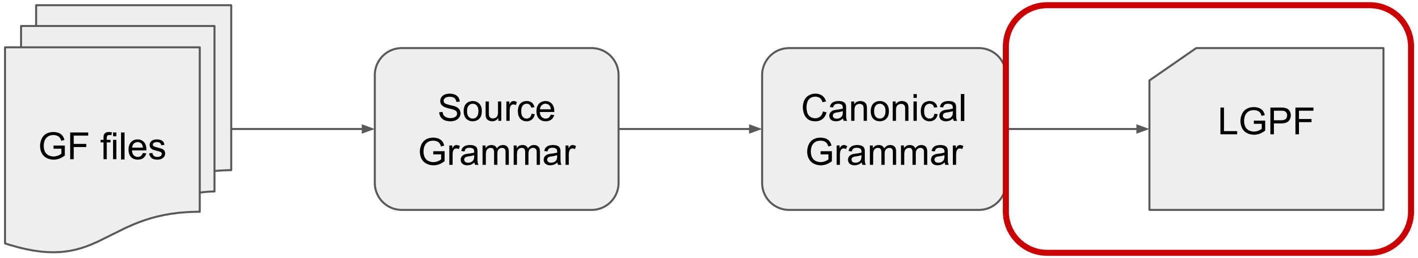
Using this should allow me to avoid the previous issues! 🤘

Hint: use `gf --output-format=canonical_gf` to see for yourselves

Using canonical format

```
grammar2canonical :: Options
                  -> ModuleName
                  -> GF.Grammar.Grammar.Grammar -- SourceGrammar
                  -> GF.Grammar.Canonical.Grammar
```

src/compiler/GF/Compile/Grammar/GrammarToCanonical.hs




weeks pass...

LPGF: Linearisation-only PGF format #103

Edit Open with

Draft johnjcamilleri wants to merge 112 commits into `master` from `lpgf`

Conversation 0 Commits 112 Checks 15 Files changed 217 +311,269 −35

johnjcamilleri commented on 17 Mar Member

Introduction

Recently I've been working on resurrecting on old idea, which is adding support for a PGF file format which *only* supports linearisation, since this is actually quite a common use for GF. The motivations are:

1. Faster & less memory-intensive compilation
2. Smaller binary files
3. Faster linearisation at runtime
4. New features impossible with parsing, e.g. dynamic lexicon.

The format itself is described in section 2 of the paper:

> "PGF: A Portable Run-Time Format for Type-Theoretical Grammars"

Reviewers

Suggestions

Thomas-H Request

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Implementation: summary

1. New output format for GF compilation:
   `gf --make --output-format=lpgf FoodsEng.gf`
2. Binary format for grammar distribution: `Foods.lpgf`
3. Haskell runtime library: `import LPGF` (next slide)
4. Testsuite for checking correctness w.r.t. treebank (testsuite/lpgf)
5. Benchmark for comparing LPGF with PGF/PGF2

Using LPGF Haskell runtime

```haskell
import LPGF
import qualified Data.Map as M

main :: IO ()
main = do
  lpgf <- readLPGF "Foods.lpgf"
  let
    Just eng = readLanguage "FoodsEng"
    Just concr = M.lookup eng (concretes lpgf)
    Just tree = readExpr "Pred (This (Mod Italian Cheese)) (Very Expensive)"
    str = linearizeConcrete concr tree
  putStrLn str
```

Two notable issues

1. Variants
2. Missing linearisation functions

Variants

```
lin car_N = mkN "Auto" "Autos" neuter
          | mkN "Wagen" "Wagen" masculine ;
```

Source: LexiconGer.gf (RGL)

- Forgivable in parsing, problematic in linearisation
- Can cause combinatorial explosions during compilation
- Somewhat debated (see issues #14, #36, #37)
- Uncontrollable variation is **not in the spirit of NLG**
- My solution: don't implement them (always pick first variant)

Missing linearisation functions

Phrasebook.gf

```
fun Indian : Citizenship ;
```

PhrasebookSpa.gf

```
-- lin Indian = … TODO
```

Expression

```
PQuestion (HowFar (ThePlace (CitRestaurant Indian)))
```

PGF

```
¿a qué distancia el restaurante [Indian] está?
```

LPGF

```
¿a qué distancia el [Indian] está?
```

benchmarks & profiling

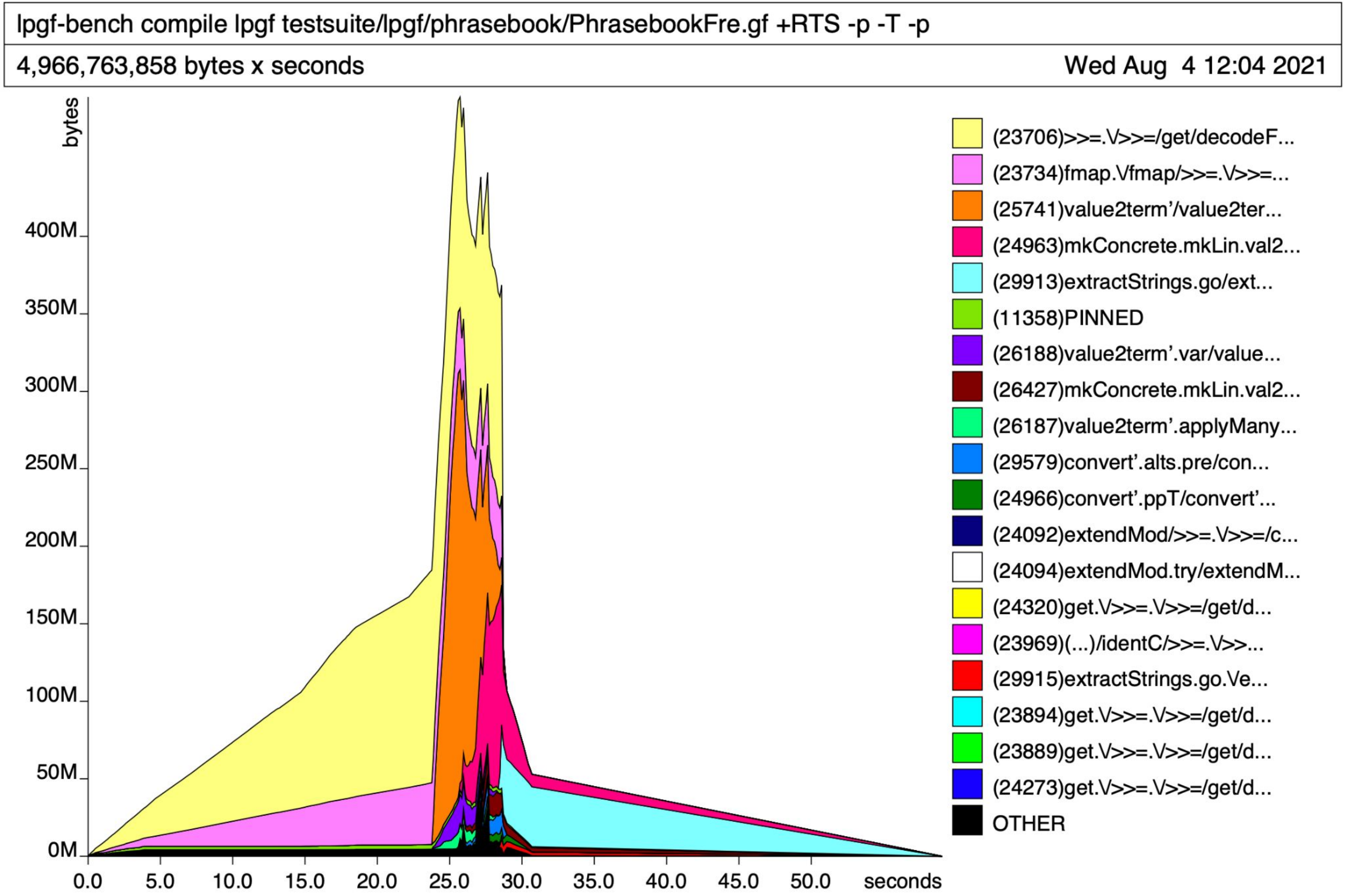
is it really as fast as we hoped?

Benchmark results

Using Phrasebook grammar in all languages (assuming `*.gfo` files precompiled)

Compilation	Time	Memory	File size
PGF	33.47s	6.42 GB	39.75 MB
LPGF	**53.60s**	**9.84 GB**	**13.12 MB**

Linearisation (10,000 trees)	Time	Memory
PGF	5.67s	2650.00 MB
PGF2 (C runtime)	50.66s	109.05 MB
LGPF	**3.49s**	**718.27 MB**


lpgf-bench compile lpgf testsuite/lpgf/phrasebook/PhrasebookFre.gf +RTS -p -T -p
4,966,763,858 bytes x seconds
Wed Aug 4 12:04 2021
bytes
400M
350M
300M
250M
200M
150M
100M
50M
0M
0.0
5.0
10.0
15.0
20.0
25.0
30.0
35.0
40.0
45.0
50.0
seconds
(23706)>>=.\>>=/get/decodeF...
(23734)fmap.\fmap/>>=.\>>=...
(25741)value2term'/value2ter...
(24963)mkConcrete.mkLin.val2...
(29913)extractStrings.go/ext...
(11358)PINNED
(26188)value2term'.var/value...
(26427)mkConcrete.mkLin.val2...
(26187)value2term'.applyMany...
(29579)convert'.alts.pre/con...
(24966)convert'.ppT/convert'...
(24092)extendMod/>>=.\>>=/c...
(24094)extendMod.try/extendM...
(24320)get.\>>=.\>>=/get/d...
(23969)(...)/identC/>>=.\>>...
(29915)extractStrings.go.Ve...
(23894)get.\>>=.\>>=/get/d...
(23889)get.\>>=.\>>=/get/d...
(24273)get.\>>=.\>>=/get/d...
OTHER

```
	Wed Aug  4 12:04 2021 Time and Allocation Profiling Report  (Final)

	   lpgf-bench +RTS -p -T -p -h -RTS compile lpgf testsuite/lpgf/phrasebook/PhrasebookFre.gf

	total time  =       22.68 secs   (22681 ticks @ 1000 us, 1 processor)
	total alloc = 11,568,701,440 bytes  (excludes profiling overheads)

COST CENTRE                  MODULE                        SRC                                                             %time %alloc

>>=.\                        Data.Binary.Get               src/runtime/haskell/Data/Binary/Get.hs:(129,28)-(130,54)         71.7    9.3
return                       Data.Binary.Get               src/runtime/haskell/Data/Binary/Get.hs:126:5-34                   1.9    3.8
fmap.\                       Data.Binary.Get               src/runtime/haskell/Data/Binary/Get.hs:(117,27)-(118,48)          1.7    5.3
value2term'                  GF.Compile.Compute.Concrete   src/compiler/GF/Compile/Compute/Concrete.hs:(504,1)-(553,30)      1.4   11.7
mkConcrete.mkLin.val2lin     GF.Compile.GrammarToLPGF      src/compiler/GF/Compile/GrammarToLPGF.hs:(123,9)-(274,71)         0.9    2.4
>>=                          GF.Data.ErrM                  src/compiler/GF/Data/ErrM.hs:(38,3)-(39,21)                       0.9    3.7
>>                           Data.Binary.Put               src/runtime/haskell/Data/Binary/Put.hs:(97,5)-(100,35)            0.9    5.2
convert'.ppT                 GF.Compile.GrammarToCanonical src/compiler/GF/Compile/GrammarToCanonical.hs:(178,5)-(203,47)    0.7    1.1
extractStrings.go.\.str      GF.Compile.GrammarToLPGF      src/compiler/GF/Compile/GrammarToLPGF.hs:345:15-38                0.6    3.9
mconcatMap                   GF.Grammar.Macros             src/compiler/GF/Grammar/Macros.hs:504:1-30                        0.5    3.1
extractStrings.go            GF.Compile.GrammarToLPGF      src/compiler/GF/Compile/GrammarToLPGF.hs:(334,5)-(363,19)         0.5    2.1
tableTypes.tabtys            GF.Compile.GrammarToCanonical src/compiler/GF/Compile/GrammarToCanonical.hs:(128,5)-(132,31)    0.5    3.4
value2term'.v2txs            GF.Compile.Compute.Concrete   src/compiler/GF/Compile/Compute/Concrete.hs:536:5-32              0.5    6.1
mkConcrete.mkLin.val2lin.grps GF.Compile.GrammarToLPGF     src/compiler/GF/Compile/GrammarToLPGF.hs:180:15-44                0.3    1.4
compare                      GF.Infra.Ident                src/compiler/GF/Infra/Ident.hs:58:17-19                           0.3    3.7
convert'.alts.pre            GF.Compile.GrammarToCanonical src/compiler/GF/Compile/GrammarToCanonical.hs:(243,9)-(247,55)    0.3    1.1
put                          Data.Binary                   src/runtime/haskell/Data/Binary.hs:318:5-22                       0.2    1.1
unzipR.(...)                 GF.Grammar.Macros             src/compiler/GF/Grammar/Macros.hs:170:35-51                       0.2    1.4
fmap                         GF.Data.ErrM                  src/compiler/GF/Data/ErrM.hs:(53,3)-(54,24)                       0.2    1.0
convert'.ppP                 GF.Compile.GrammarToCanonical src/compiler/GF/Compile/GrammarToCanonical.hs:(218,5)-(233,49)    0.2    1.5
mkConcrete.mkLin.val2lin.\   GF.Compile.GrammarToLPGF      src/compiler/GF/Compile/GrammarToLPGF.hs:(242,47)-(244,43)        0.2    1.7
zipAssign                    GF.Grammar.Macros             src/compiler/GF/Grammar/Macros.hs:182:1-51                        0.2    1.3
ident2utf8                   GF.Infra.Ident                src/compiler/GF/Infra/Ident.hs:(83,1)-(88,16)                     0.2    1.2
convert'.ppP.fields          GF.Compile.GrammarToCanonical src/compiler/GF/Compile/GrammarToCanonical.hs:232:9-57            0.2    1.2
unzipR                       GF.Grammar.Macros             src/compiler/GF/Grammar/Macros.hs:170:1-51                        0.1    1.1
```

Optimisations

In place ✅

- Pruning impossible subterms
- Raw strings stored exactly once in binary file
- Make `GF.Grammar.Canonical.Id` a type synonym for `GF.Infra.Ident.RawIdent`

Tested without success ❌

- Use `String` instead of `Data.Text` ([`lpgf-string`])
- Memoisation of subterms ([`lpgf-memo`])

Untested ❔

- Playing with Haskell laziness
- Optimising `extractStrings` optimisation

conclusions

Status summary

- It works*!
- LPGF files are [often] smaller than PGFs 😄
- Runtime linearisation in LPGF is faster than both PGF and PGF2 🥳
- Compiling to LPGF is at least as slow/memory-consuming as PGF, often significantly worse 😢
- Extra features… 

* except for variants and missing functions

Roadmap

Immediate priorities

- As fast and small as possible
  - compilation
  - runtime (is Haskell a problem here?)
- More features to runtime API
  - type-checking of user-generated expressions

Probably also want

- GF shell support
- Runtime in other languages (native or via bindings)
  - JavaScript?
  - Python?
  - Java?
  - C?

Extra features

1. Dynamic lexicon
    - No recompiling when lexicon changes
    - Linearise function in runtime takes extra `Lexicon` argument
    - But: access to smart paradigms?
    - Syncing lexicon with grammar
2. No need to know all tokens at compile time
    - it could allow things such as runtime gluing
    - conversion of integers to numeral expressions, which is often expected by users
    - lin-only extensions of the GF source language
3. Multi-PGF format
    - some concrete syntaxes are full PMCFG
    - some concrete syntaxes are linearisation-only PGF
    - parsing with a linearisation-only language is an error
    - all other cases work seamlessly

text:ual

https://textual.ai/

- Presented at GFSS 2018, South Africa
- Pure NLG for product descriptions in e-commerce
- 25 languages
- Lexicon is huge and constantly updated
- Recompiling to PGF on-the-fly is impractical
- Home-grown dynamic lexicon solution using placeholders

Learn more & contribute

- Questions to the community
  - Who else is this valuable for?
  - Which features do you want to see/prioritise?
- Draft pull request
  - https://github.com/GrammaticalFramework/gf-core/pull/103
  - Should we merge it?
- LPGF Dev README
  - https://github.com/GrammaticalFramework/gf-core/blob/lpgf/testsuite/lpgf/README.md
  - How to run tests/benchmarks
  - Details from various investigations
- Help with optimisations

Credits

- Aarne, for reviving this idea
- Krasimir, who knows everything about PGF
- Andreas "Anka" Källberg, for helping with Haskell profiling