A reference GLL implementation

October 2023
Adrian Johnstone

ROYAL HOLLOWAY UNIVERSITY OF LONDON

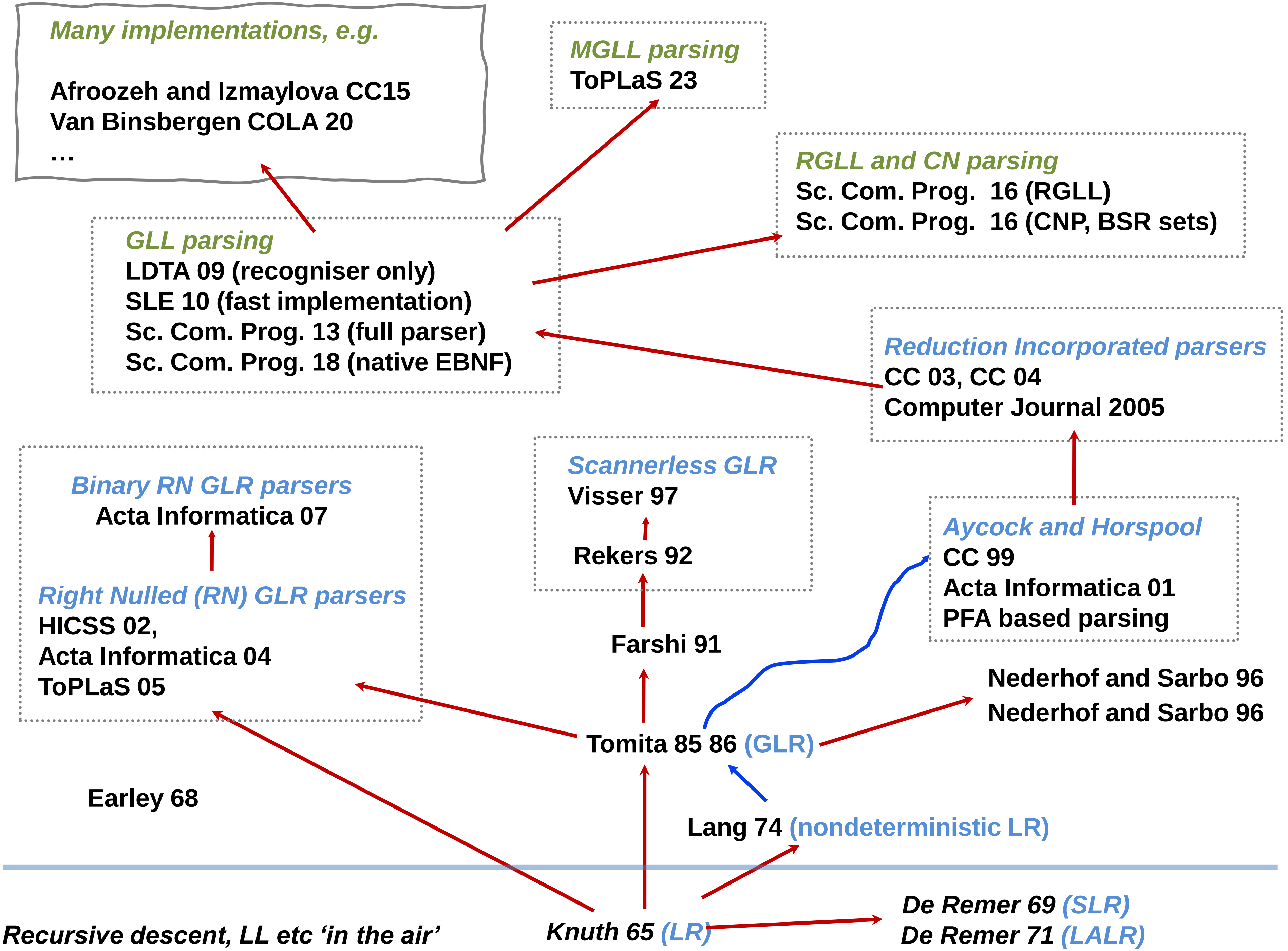
Acknowledgements

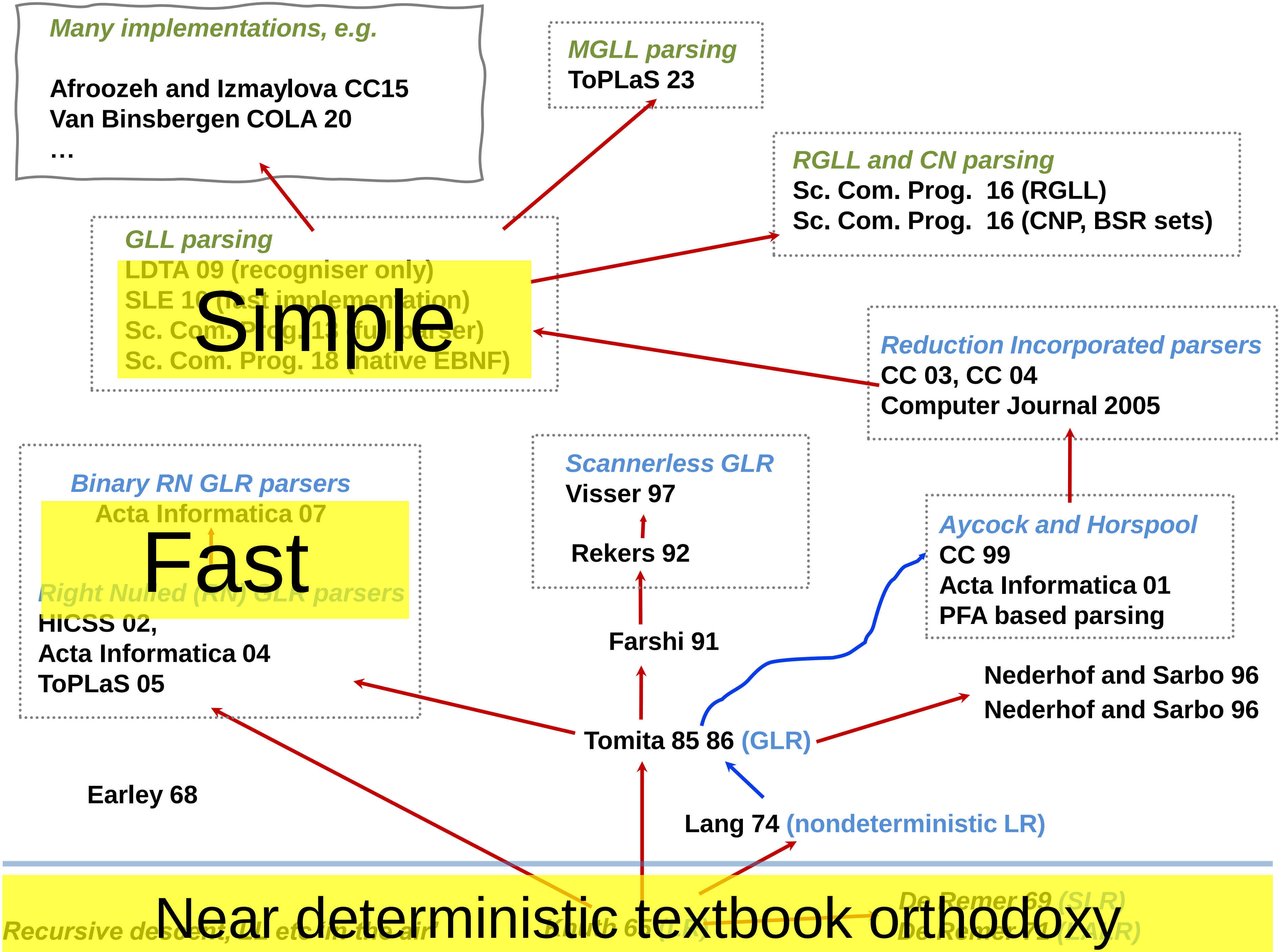
The Leverhulme Trust

EPSRC
Engineering and Physical Sciences
Research Council

The Leverhulme Trust for project grant RPG-2013-396

EPSRC for project EP/I032509/1


Many implementations, e.g.
Afroozeh and Izmaylova CC15
Van Binsbergen COLA 20
…
MGLL parsing
ToPLaS 23
RGLL and CN parsing
Sc. Com. Prog. 16 (RGLL)
Sc. Com. Prog. 16 (CNP, BSR sets)
GLL parsing
LDTA 09 (recogniser only)
SLE 10 (fast implementation)
Sc. Com. Prog. 13 (full parser)
Sc. Com. Prog. 18 (native EBNF)
Reduction Incorporated parsers
CC 03, CC 04
Computer Journal 2005
Binary RN GLR parsers
Acta Informatica 07
Right Nulled (RN) GLR parsers
HICSS 02,
Acta Informatica 04
ToPLaS 05
Scannerless GLR
Visser 97
Rekers 92
Farshi 91
Aycock and Horspool
CC 99
Acta Informatica 01
PFA based parsing
Nederhof and Sarbo 96
Nederhof and Sarbo 96
Tomita 85 86 (GLR)
Earley 68
Lang 74 (nondeterministic LR)
De Remer 69 (SLR)
De Remer 71 (LALR)
Recursive descent, LL etc 'in the air'
Knuth 65 (LR)


Many implementations, e.g.
Afroozeh and Izmaylova CC15
Van Binsbergen COLA 20
…
MGLL parsing
ToPLaS 23
RGLL and CN parsing
Sc. Com. Prog. 16 (RGLL)
Sc. Com. Prog. 16 (CNP, BSR sets)
GLL parsing
LDTA 09 (recogniser only)
SLE 10 (fast implementation)
Sc. Com. Prog. 13 (full parser)
Sc. Com. Prog. 18 (native EBNF)
Simple
Reduction Incorporated parsers
CC 03, CC 04
Computer Journal 2005
Binary RN GLR parsers
Acta Informatica 07
Fast
Right Nulled (RN) GLR parsers
HICSS 02,
Acta Informatica 04
ToPLaS 05
Scannerless GLR
Visser 97
Rekers 92
Farshi 91
Aycock and Horspool
CC 99
Acta Informatica 01
PFA based parsing
Nederhof and Sarbo 96
Nederhof and Sarbo 96
Tomita 85 86 (GLR)
Earley 68
Lang 74 (nondeterministic LR)
De Remer 69 (SLR)
Recursive descent, LL etc 'in the air'
Knuth 65 (LR)
De Remer 71 (LALR)
Near deterministic textbook orthodoxy

LALR/LL parsing: fine for linear things
Flips over if you drive it too hard

Santa Pod European Final 2010

Singleton derivation parsers: OK on special surfaces
Always follows one track, ignoring other routes

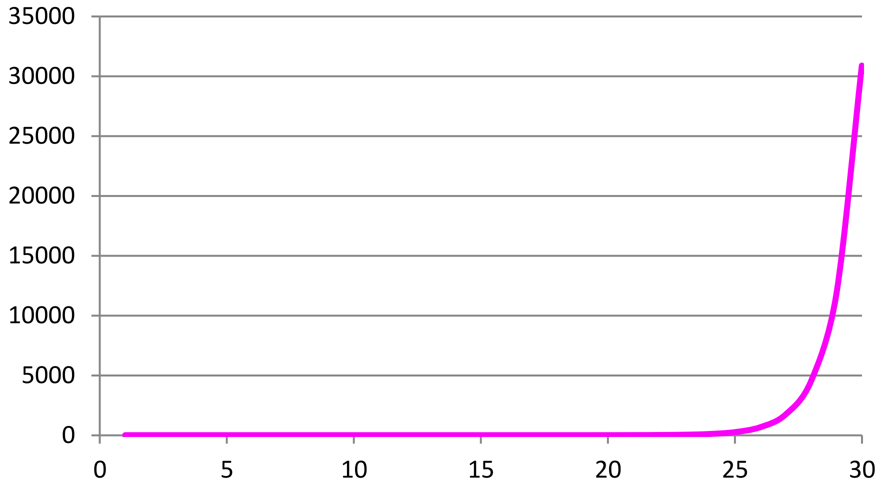
2011 Japanese GP: Jenson Button (McLaren) during race

General parsers: go anywhere
Slow and bulky

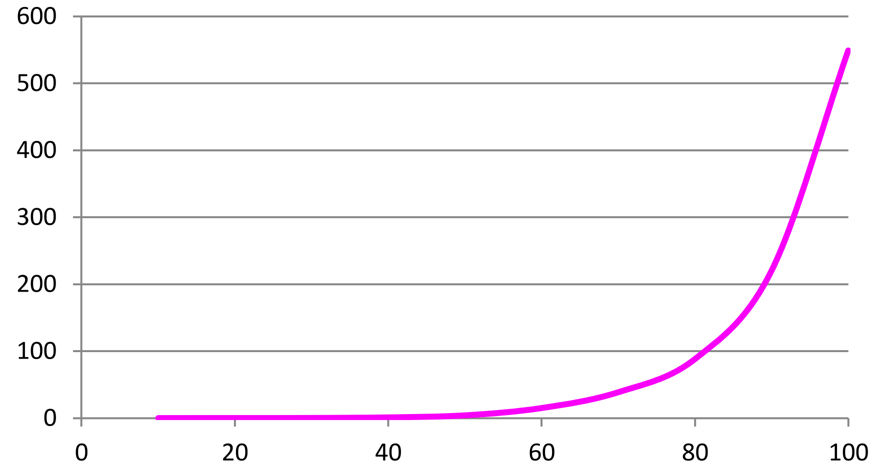
Land Rover Defender 90 1999

https://www.youtube.com/watch?app=desktop&v=Cz1BpbsbFkA

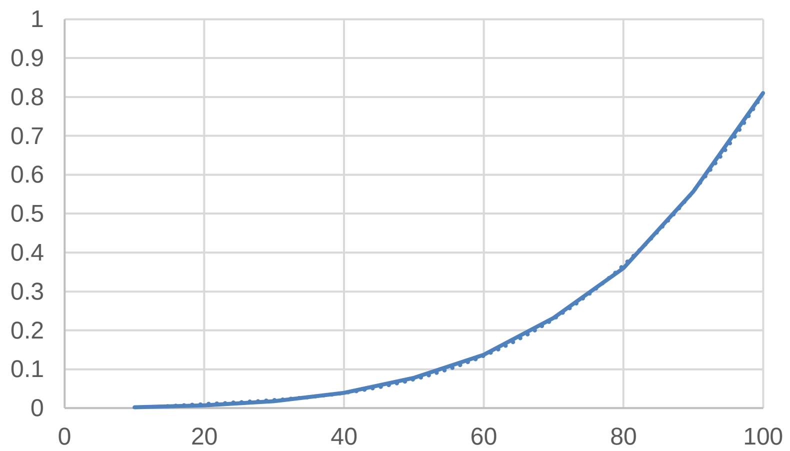
Worst case: parse b^n with S ::= b | S S | S S S



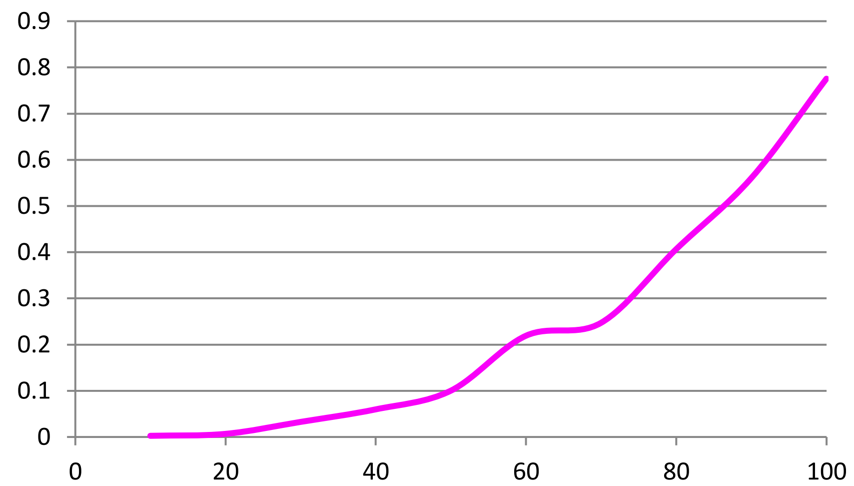
JSDF (2013)



SDF2 (2013)



GLLHP (Java, 2023)



ARTV3 compiled GLL (ANSI C, 2013)

Paper overview

Notation and compiled vs interpreted parsers
Grammar representation
Parser context

Backtracking recursive descent (neophyte friendly)
From compiled to state based interpretation

Baseline – gllBL (Java API user friendly)

Memory management and hash pools
Memory efficient baseline – gllHP (hash pool)

Variants to be characterised

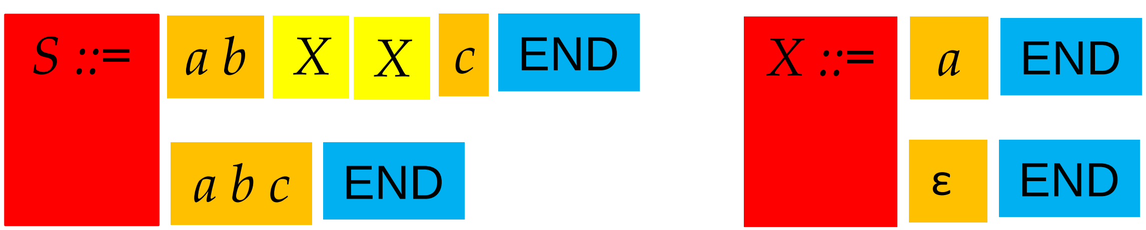
GLL control flow fragments

A CFG is a nondeterministic specification of a language

GLL provides one style of sequentialisation for the general parsing problem

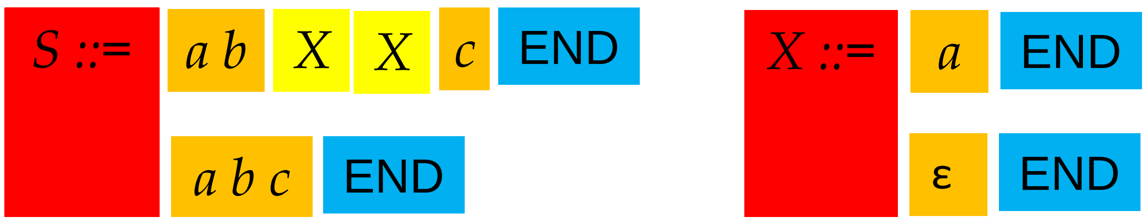
The basic idea is to split the grammar specification up into a set of GLL fragments roughly corresponding to the basic blocks in a recursive descent parser

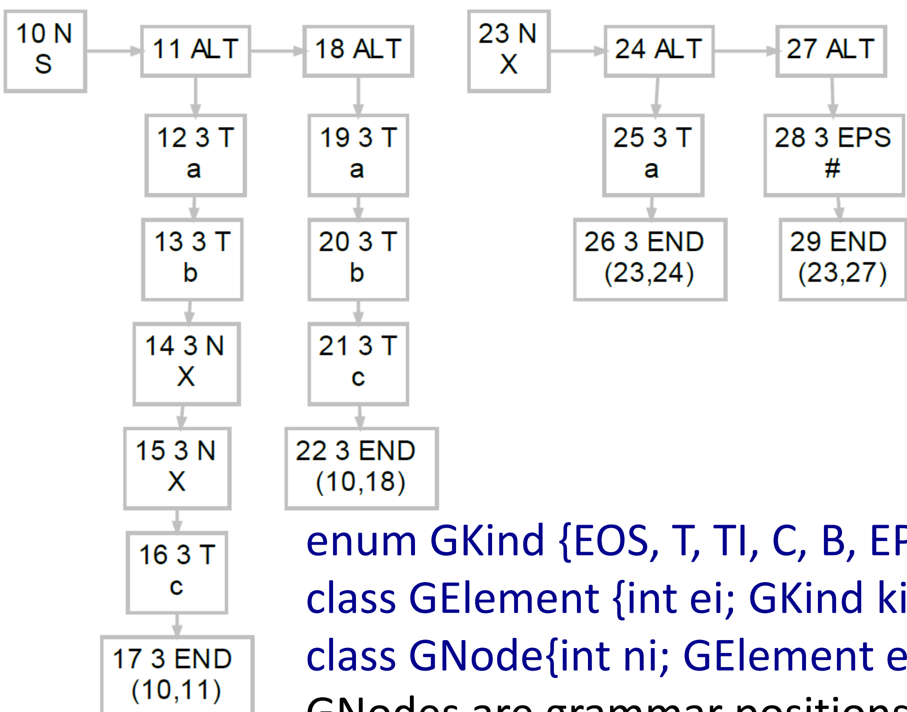
S ::= a b X X c | a b c   X ::= a | #



Interpreter friendly grammar rep

S ::= a b X X c | a b c   X ::= a | #





End.seq references start of this production
END.alt references start of production list

Extends naturally to EBNF brackets
if END.alt references the enclosing bracket

enum GKind {EOS, T, TI, C, B, EPS, N, ALT, END, DO, OPT, POS, KLN}
class GElement {int ei; GKind kind; String str;}
class GNode{int ni; GElement el; GNode alt, seq;}
GNodes are grammar positions – *slots* in classical GLL

The six sets

Threads
Thread descriptors

GLL BL declarations

class Descriptor {GNode gn; int i; SNode sn; DNode dn;}

Stack
GSS nodes
GSS edges
Pops

class GSSN {GNode gn; int i; Set<GSSE> edges;
Set<SPPFN> pops;}
class GSSE {GSSN dst; SPPFN sppfnode; }

Pops are distributed over the GSS nodes
(Set of pairs in classical GLL)

Derivations
SPPF nodes
SPPF packed nodes

class SPPFN {GNode gn; int li; int ri; Set<SPPFPN> packNS;}

class SPPFPN {GNode gn; int pivot; SPPFN lC; SPPFN rC;}

Baseline gll

```
void gllBL() {
 initialise();
 nextDescriptor: while (dequeueDesc())
 while (true) {
  switch (gn.elm.kind) {
   case B,T,TI,C: if (input[i] == gn.elm.ei)
           {du(1); i++; gn = gn.seq; break;}
           else continue nextDescriptor;
   case N: call(gn); continue nextDescriptor;
   case EPS: du(0); gn = gn.seq; break;
   case END: ret(); continue nextDescriptor;

   case DO: gn = gn.alt; break;
   case ALT:
     for (GNode tmp = gn; tmp != null; tmp = tmp.alt)
       queueDesc(tmp.seq, i, sn, dn);
     continue nextDescriptor;
}}}
```

Control flow sequentialisation

```
void call(GNode gn) {
 GSSN gssN = gssFind(gn.seq, i);
 GSSE gssE = new GSSE(sn, dn);

 if (!gssN.edges.contains(gssE)) {
  gssN.edges.add(gssE);
  for (SPPFN rc : gssN.pops)
   queueDesc(gn.seq, rc.ri, sn,
        sppfUpdate(gn.seq, dn, rc));
 }

 for (GNode p = rules(gn).alt;
      p != null; p = p.alt)
  queueDesc(p.seq, i, gssN, null);
}
```

```
void ret() {
 if (sn.equals(gssRoot)) { // Stack base
  if (accepting(gn) &&
     (i == input.length - 1)) {
    sppfRootNode =
    sppf.get(new SPPFN(
    startNonterminal, 0, input.length - 1));
    accepted = true;
  } else
    rightmostParseIndex = sppfWidestIndex();
  return; // End of parse
 }
 sn.pops.add(dn);
 for (GSSE e : sn.edges)
  queueDesc(sn.gn, i, e.dst,
         sppfUpdate(sn.gn, e.sppfnode, dn));
}
```

Control flow sequentialisation

```
void call(GNode gn) {
 GSSN gssN = gssFind(gn.seq, i);
 GSSE gssE = new GSSE(sn, dn);

 if (!gssN.edges.contains(gssE)) {
  gssN.edges.add(gssE);
  for (SPPFN rc : gssN.pops)
   queueDesc(gn.seq, rc.ri, sn,
         sppfUpdate(gn.seq, dn, rc));
 }

 for (GNode p = rules(gn).alt;
      p != null; p = p.alt)
  queueDesc(p.seq, i, gssN, null);
}
```

```
void ret() {
 if (sn.equals(gssRoot)) { // Stack base
  if (accepting(gn) &&
     (i == input.length - 1)) {
    sppfRootNode =
    sppf.get(new SPPFN(
    startNonterminal, 0, input.length - 1));
    accepted = true;
  } else
    rightmostParseIndex = sppfWidestIndex();
  return; // End of parse
 }
 sn.pops.add(dn);
 for (GSSE e : sn.edges)
  queueDesc(sn.gn, i, e.dst,
         sppfUpdate(sn.gn, e.sppfnode, dn));
}
```

Memory management

Allocate pool blocks using Java runtime

One hash table per set; hand-code offsets to set element fields

No safety net, and anathema to OO programmers

```
/*Constant field offsets. Offset zero links to next in hash chain */
 protected final int gssNode_gn = 1; // Key
 protected final int gssNode_i = 2; // Key
 protected final int gssNode_edgeList = 3;
 protected final int gssNode_popList = 4;
 protected final int gssNode_SIZE = 5; // Key size 2

 protected final int gssEdge_src = 1; // Key
 protected final int gssEdge_dst = 2; // Key
 ...

/* Lookup key <a,b>
   If not found, allocate allocationSize and load <a,b> to offsets (1,2) */
 protected void find(int[] hashBuckets, int hashBucketCount, int allocationSize,
                     int a, int b) {
   hash(hashBucketCount, a, b);

   findIndex = hashBuckets[hashResult];
   do {
     findBlockIndex = findIndex >> poolAddressOffset;
     findOffset = findIndex & poolAddressMask;
   ...
```

Characterising the variants

Target: fast BRNGLR parsing (ToPLaS05, Acta Inf.07)

Parenthesised BNF and FBNF (SLE 10)
Lookahead (SCP 13)
EBNF (SCP 18)
Generated parsers: ANSI C; Java fragmentation issue
RGLL (SCP 16)
CNP (SCP 16)
BSR sets (SCP 16)
MGLL and multiparsing (ToPLaS23)

Good enough for GNU?

GEG - no more than 10% slow down against g++ -Ofast
gcc parser went hand-crafted around 2004

One (large - 117 Kbyte) example:

g++	18.8s
gllHP on ANSI-C	1.35s
gllHP on ANSI-C++	7.02s

Now add these optimisations:

- Lookahead and iteration for EBNF
- BSR sets instead of full SPPF representation
- Generated C parser

Repos

Code and small examples from the paper
https://github.com/AJohnstone2007/referenceImplementation

Java 18 and SML grammars with large test sets (from ToPLaS23)
https://github.com/AJohnstone2007/referenceLanguageCorpora

Research papers mentioned on slide 3
https://pure.royalholloway.ac.uk/en/persons/adrian-johnstone/publications

Future investigations

- Linear handling of deterministic sub languages
  - (cf Scott McPeak CC04 – GLR/LR parser)
- Threaded multicore
- Threaded