Lexerless GLR and Boolean Grammars

Adam Megacz

02-Nov-2005

JBP: Java Boolean Parser

(for lack of a better name)

* Motivation: need GLR implementation that is:
  * Lexerless
  * In Java (emits parser as Java source code)
* Bonus features:
  * Programmatic grammar manipulation
  * Boolean Grammars (superset of Context-Free)

Outline

1. LR Parsing
2. GLR Parsing
3. History of GLR
4. Lexerless Parsing
5. Conjunctive Grammars
6. Boolean Grammars
7. Other features

LR Parsing

* "Left-to-Right": on-line parsing
* O(n) time
* Only works for "follow-deterministic" grammars

LR Parsing

```
Expr ::= Expr “+” Expr
       | Expr “-” Expr
       | “(” Expr “)”
       | [0-9]+
```

LR Parsing

```
Expr ::= Expr “+” Expr
       | Expr “-” Expr
       | “(” Expr “)”
       | [0-9]+
```

Input: “(1+2)-3”

LR Parsing

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | "(" Expr ")"
       | [0-9]+
```

Input: "(1+2)-3"

Shift "("

(

LR Parsing

```
Expr ::= Expr “+” Expr
       | Expr “-” Expr
       | “(” Expr “)”
       | [0-9]+
```

Input: “(1+2)-3”

Shift “1”

1

(

LR Parsing

1

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | "(" Expr ")"
       | [0-9]+
```

Input: "(1+2)-3"

Shift "+"

+

1

(

1

LR Parsing

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | "(" Expr ")"
       | [0-9]+
```

Input: "(1+2)-3"

Shift "2"

2

+

1

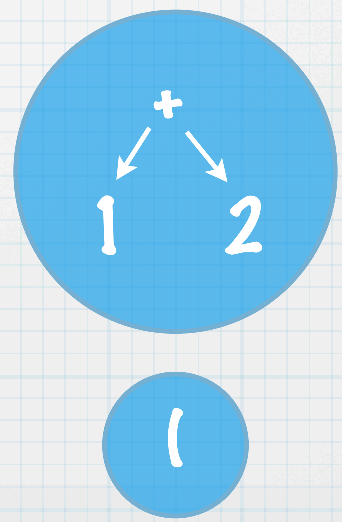
(

LR Parsing

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | "(" Expr ")"
       | [0-9]+
```

```
Input: "(1+2)-3"
```

Reduce "Expr + Expr"



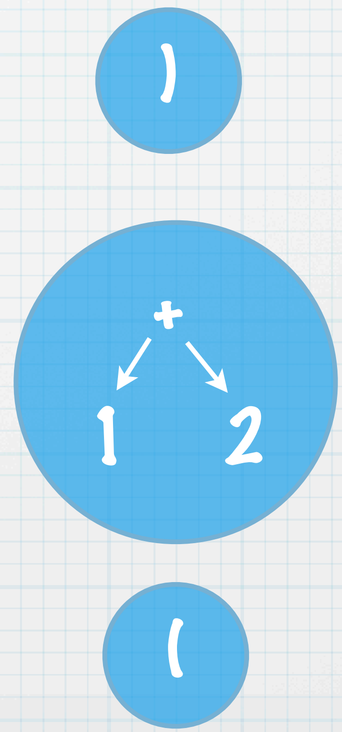
1

LR Parsing

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | "(" Expr ")"
       | [0-9]+
```

Input: "(1+2)-3"

Shift ")"

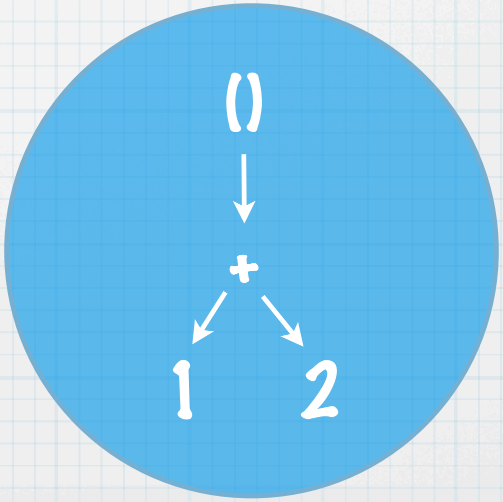


LR Parsing

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | "(" Expr ")"
       | [0-9]+
```

Input: "(1+2)-3"

Reduce "(Expr)"

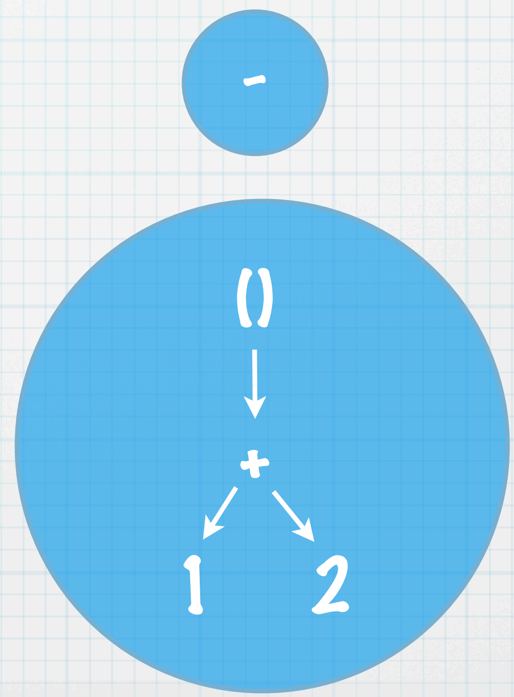


LR Parsing

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | "(" Expr ")"
       | [0-9]+
```

Input: "(1+2)-3"

Shift "-"

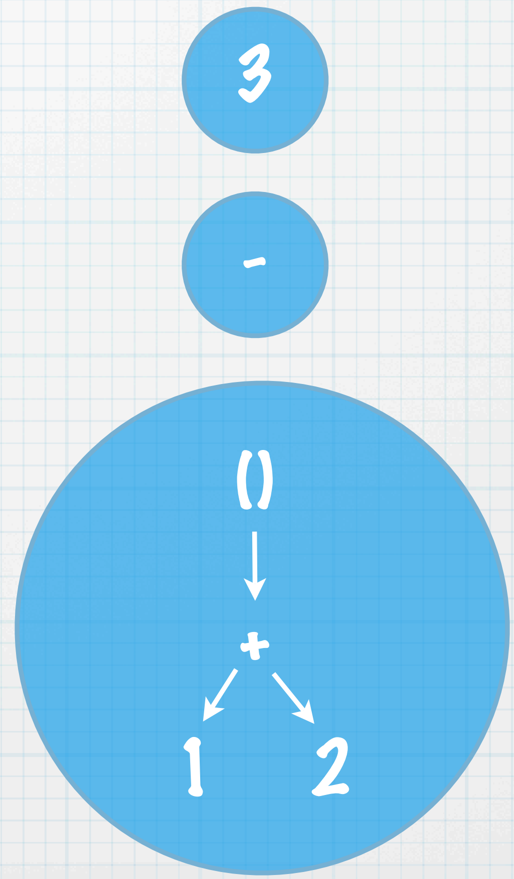


LR Parsing

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | "(" Expr ")"
       | [0-9]+
```

Input: "(1+2)-3"

Shift "3"

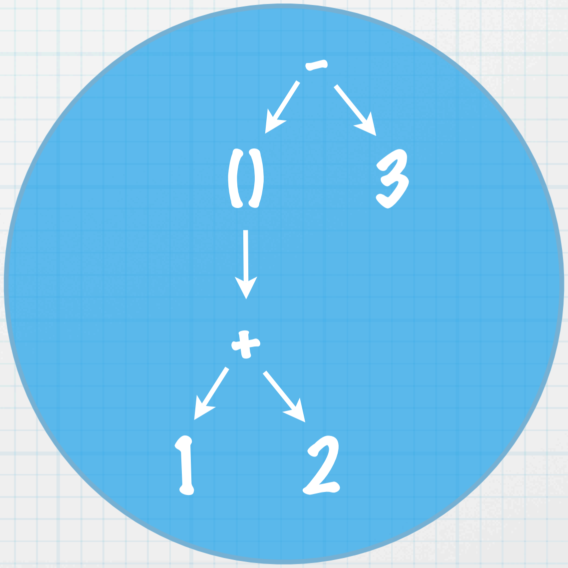


LR Parsing

```
Expr ::= Expr "+" Expr
       | Expr "-" Expr
       | "(" Expr ")"
       | [0-9]+
```

Input: "(1+2)-3"

Reduce "Expr - Expr"



Key LR Invariant

* The nodes along the path from the top of the stack to the bottom represent parse tree fragments for elements of a prefix chain of productions

```
Expr ::= "(" Expr ")"
       | Expr "+" Expr
       | [0-9]+
```

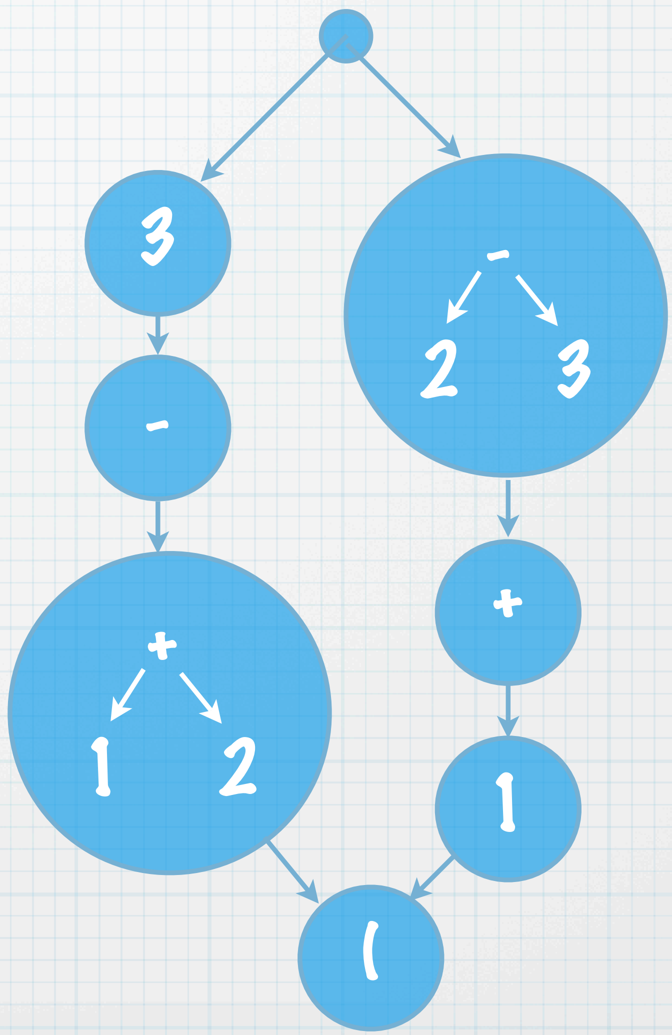
Input: "(1+2)-3"

GLR (Generalized LR)

- Not only for "follow-deterministic" grammars, but $O(n)$ on them like LR
- $O(n^3)$ worst case
  - Almost always avoidable
- Three key concepts
  - Multiple reduction paths
  - Graph Structured Stack
  - Shared, Packed Parse Forest

GLR (Generalized LR)

* The nodes along **each path** from the top of the **graph structured stack (GSS)** to the bottom represent **shared packed parse forest (SPPF)** fragments for elements of a prefix chain of productions

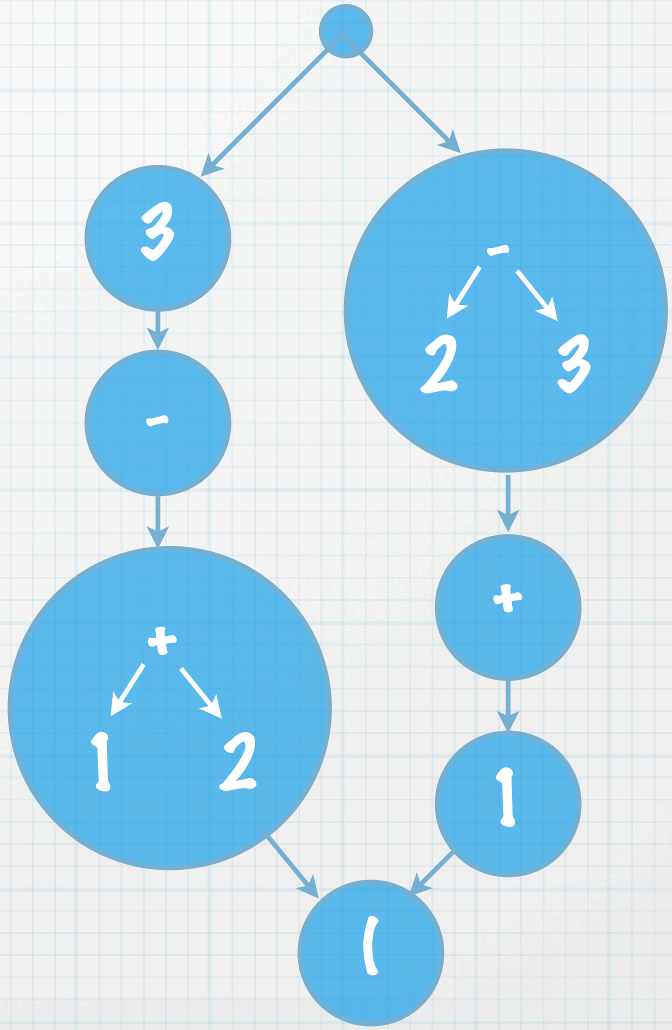


Input: "(1+2-3)"

Multiple Paths

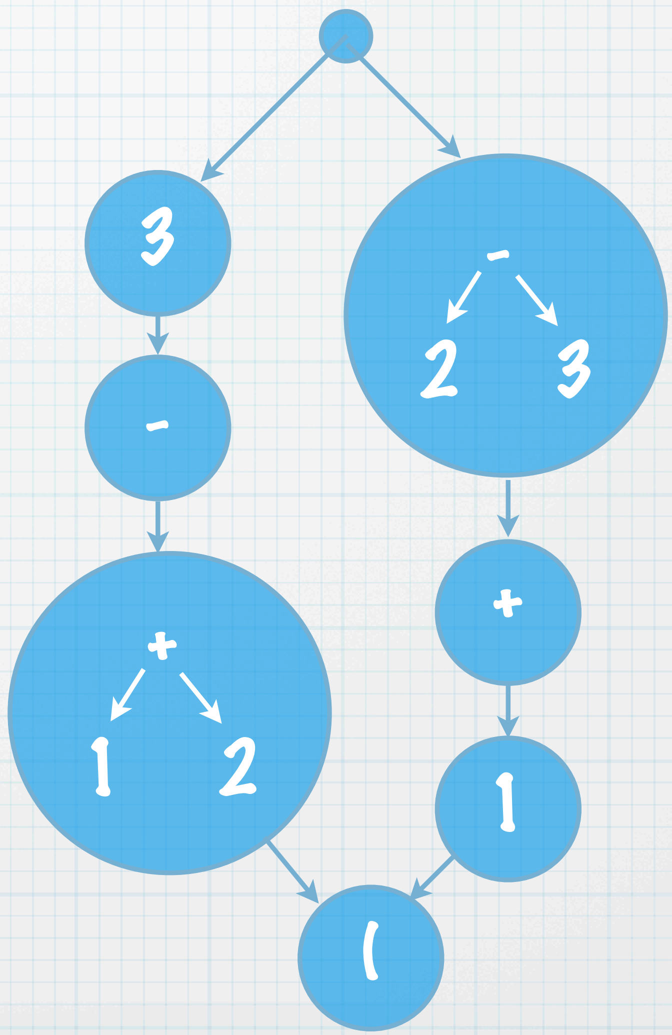
* The nodes along **each path** from the top of the GSS to the bottom represents SPPF fragments for elements of a prefix chain of productions

* **Multiple paths indicate multiple possible prefix chains**



Graph Structured Stack

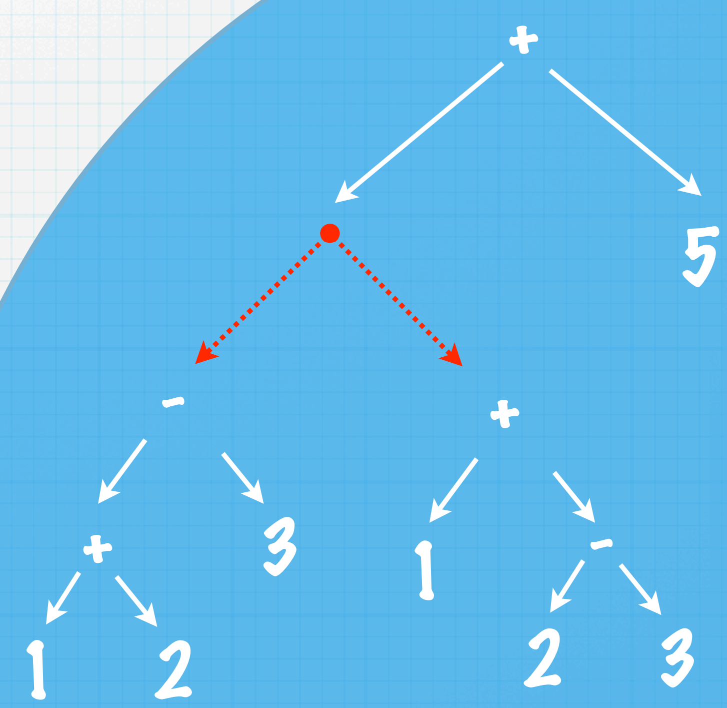
* The nodes along each path from the top of the GSS to the bottom represents SPPF fragments for elements of a prefix chain of productions
* Graph structured stack allows sharing; bounds graph at O(n^2) (for CNF grammars)



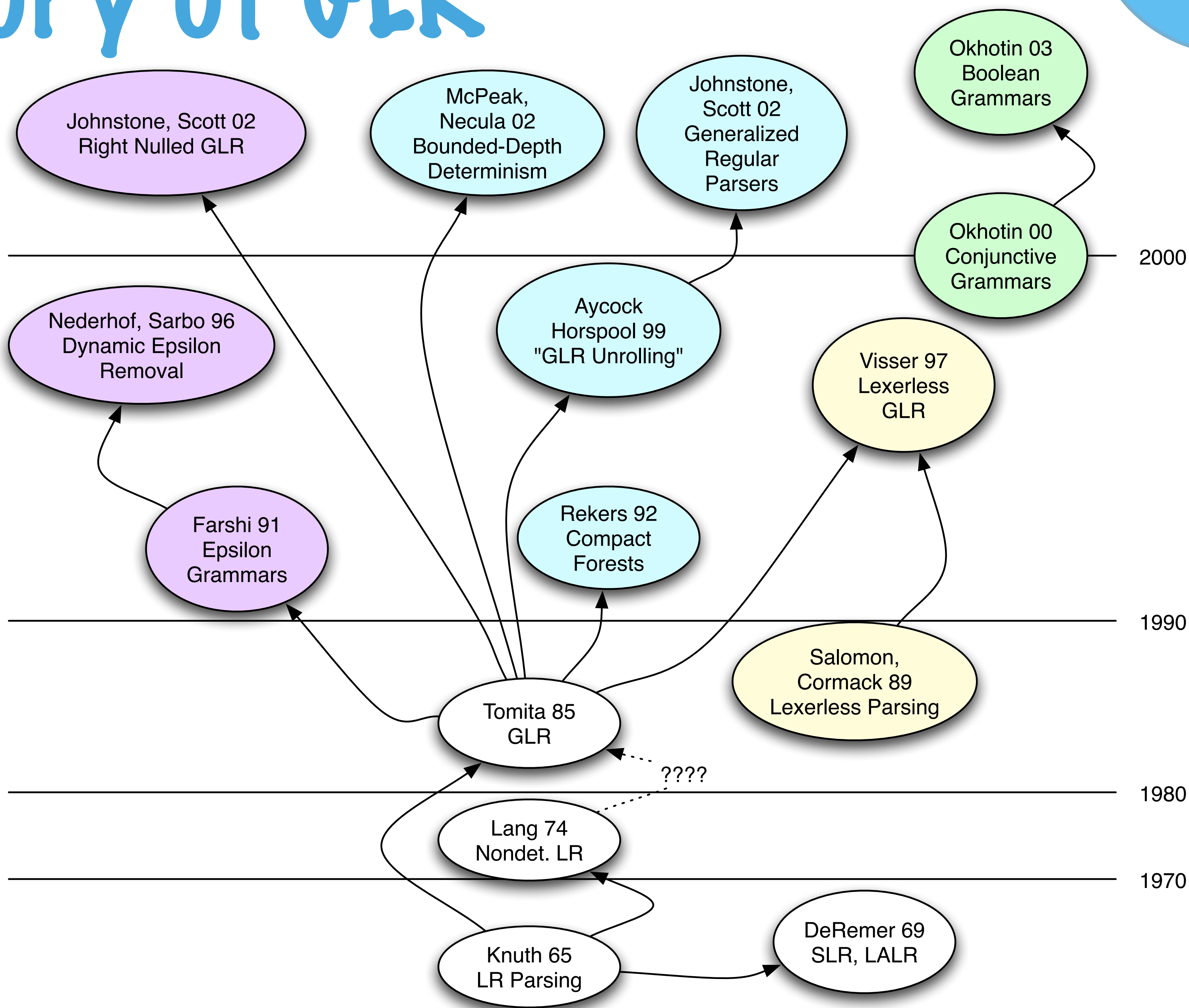
Shared Packed Parse Forest

* The nodes along each path from the top of the GSS to the bottom represents **SPPF** fragments for elements of a prefix chain of productions
* **Shared, Packed Parse Forest represents multiple valid parse trees efficiently**

Input: (1+2-3)+5



History of GLR



Lexerless GLR

* No lexer; every character is a parse token
* Also called "scannerless" and "complete character-level"
* Advantage: single formalism for entire syntax
* Disadvantage: requires strange "features"
    * Follow
    * Reject
    * Prefer/Avoid
    * Character Ranges
    * Whitespace insertion

Lexerless GLR extensions

* Character Ranges
  * Specify A-Z without typing 26 things
  * Specify "all non-whitespace unicode chars" without typing 65,000-some things
  * A robust implementation needs to use range-set arithmetic when constructing the parse table

Lexerless GLR extensions

- Follow Restrictions
  - simulate longest-match token

```
Identifier ::= [A-Za-z]+ ~/~ [A-Za-z]
```

↑ "not followed by" (negative lookahead)

Lexerless GLR extensions

* Reject Productions
    * identifiers cannot be keywords
    * interesting things happen when you omit this...

```
Identifier ::= [A-Za-z]+ ~/~ [A-Za-z]
             | “while” {reject}
```

reject attribute
(regardless of other productions, Identifier cannot match the text “while”)

Lexerless GLR extensions

* **Prefer/Avoid**
  * used to handle "dangling else"
  * implemented as a filter applied to the packed forest after parsing

```
S ::= "if" E "then" S
    | "if" E "then" S "else" S
```

↑ must not end with an if..then

Conjunctive Grammars

* Straightforward concept, but details not worked out until Okhotin '00
* In addition to juxtaposition and union, allows intersection (&) as an operator in grammar productions

Conjunctive Grammars

```
S ::= AB & DC
A ::= "a" A         | ϵ
B ::= "b" B "c"     | ϵ
C ::= "c" C         | ϵ
D ::= "a" D "b"     | ϵ
```

$\{ a^n b^n c^n \}$

Not context-free!

Conjunctive Grammars

* Despite added power, still parseable in O(n^3)
* GLR algorithm already gives us most of what we need
  * Just add the ability for two ambiguous parsings to be interdependent
    * A link between two reduction nodes
    * If either node is rejected, the other dies

Boolean Grammars

* Okhotin '03
* Adds negation to conjunctive grammars
* Raises some theoretical issues:

```
Y ::= (~Y) & X
```

* Okhotin arrives at a minimal restriction to retain sanity

Boolean Grammars

* Still parseable using GLR
  * Link between two ambiguity nodes
  * If nodes are ever merged and the negated one has not yet failed, then both must fail

* Boolean closure of follow-deterministic grammars is still parseable in linear time!

Boolean Grammars

- Why would you want to use these?
  - Okhotin uses them to make "variables must be used within scope" a syntactic constraint
  - Dirty little secret is that the resulting grammar is pathologically nondeterminitic; worst-case GLR performance

Boolean Grammars

* Better reason to use them:
  * Clean formalism, well understood
  * Subsumes most of the "ugly hacks" needed for lexerless parsing
    * Visser's algorithm for reject constraints is a special case of Okhotin's negation rule
    * Can handle dangling-else elegantly (no need for prefer/avoid constraints)

Boolean & Lexerless

* Lexerless parsing and Boolean grammars go well together
    * Cleaner formalism for Lexerless Parsing
    * Realistic application for Boolean Grammars
* Boolean grammars are just plain cool
    * Lots left to be discovered

Other Features

* Any topological space (union, intersection, complement, empty set, universe) can be used as an alphabet ("character set")
  * No assumption of a bijection with integers
  * No assumption that a bit-set is a practical representation
  * Parsing a discrete sequence of objects drawn from a non-discrete space

Other Features

* Nice API, programmatic manipulations
* All grammatical elements extend `Element`

```
Union    expr = new Union();
Element  id   = new Range('A', 'Z').many1().maximal();

expr.add(new Sequence(new Object[] { expr, "+", expr }));
expr.add(new Sequence(new Object[] { expr, "*", expr }));
expr.add(new Sequence(new Object[] { id }));
```

Other Features

* Union **implements** Collection<Sequence>
* Sequence **implements** Collection<Element>

```
Union    expr = new Union();
Element  id   = new Range('A', 'Z').many1().maximal();

expr.add(new Sequence(new Object[] { expr, "+", expr }));
expr.add(new Sequence(new Object[] { expr, "*", expr }));
expr.add(new Sequence(new Object[] { id }));

for(Sequence sequence : expr)
  for(Element element : sequence)
    System.out.print(element + " ");
```

Implementation

http://www.cs.berkeley.edu/~megacz/jbp/