Tools and Analyses for Ambiguous Input Streams

Andrew Begel and Susan L. Graham
University of California, Berkeley
LDTA Workshop - April 3, 2004

HARMONIA
http://harmonia.cs.berkeley.edu
Harmonia: Language-aware Editing

- Programming by Voice
  - Code dictation
  - Voice-based editing commands

- Program Transformations
  - Transformation actions
  - Pattern-matching constructs
Harmonia: Language-aware Editing

- Programming by Voice
  - Code dictation
  - Voice-based editing commands

- Program Transformations
  - Transformation actions
  - Pattern-matching constructs
Harmonia: Language-aware Editing

- Programming by Voice
  - Code dictation
  - Voice-based editing commands

- Program Transformations
  - Transformation actions
  - Pattern-matching constructs
Harmonia: Language-aware Editing

- Programming by Voice
  - Code dictation
  - Voice-based editing commands
- Program Transformations
  - Transformation actions
  - Pattern-matching constructs

Each kind of input stream ambiguity requires new language analyses
Speech Example

for int i equals zero i less than ten i plus plus
Ambiguities

for (int i = 0; i < 10; i++) {
    int eye equals 0 aye less then 10 i plus plus
}

for (int i = 0; i < 10; i++) {
    
}

April 3, 2004
Ambiguities

4 int eye equals 0 aye less then 10 i plus plus

for (int i = 0; i < 10; i++) {

}
Another Utterance

for times ate equals zero two plus equals one
Many Valid Parses!

for times ate equals zero two plus equals one

for (times; ate == 0; to += 1) {
    |
}

down 4 * 8 = zero; to += won

down fore.times(8).equalsZero(2, plus == 1) ▭
Embedded Language Example

- C and Regexps embedded in Flex

**Flex Rule for Identifiers**

```flex
[_a-zA-Z]([_a-zA-Z0-9]*)
i++; RETURN_TOKEN(ID);
```
Embedded Language Example

- C and Regexps embedded in Flex

**Flex Rule for Identifiers**

```
/_a-zA-Z/(_a-zA-Z0-9\])* i++; RETURN_TOKEN(ID);
```

- Why not this interpretation?

```
/_a-zA-Z/(_a-zA-Z0-9\])* i++; RETURN_TOKEN(ID);
```
Legacy Language Example

- Fortran

```
DO 57 I = 3, 10
```
Legacy Language Example

- Fortran
  - Do Loop
    DO 57 I = 3,10
Legacy Language Example

- **Fortran**

  - Do Loop
    
    ```fortran
    DO 57 I = 3, 10
    DO 57 I = 3
    ```
Legacy Language Example

- Fortran
  - Do Loop
    ```
    DO 57 I = 3,10
    ```
  - Assignment
    ```
    DO 57 I = 3
    ```
Legacy Language Example

• Fortran

  • Do Loop
    DO 57 I = 3,10
  • Assignment
    DO57I = 3
Legacy Language Example

- **PL/I**
  - Non-reserved Keywords

  ```plaintext
  IF IF = THEN
  THEN THEN = ELSE
  ELSE ELSE = END
  END
  ```
Legacy Language Example

- PL/I

  - Non-reserved Keywords

  
  ```
  IF IF = THEN
  THEN THEN = ELSE
  ELSE ELSE = END
  END
  ```
Input Stream Classification

- Single Lexical Category
  - Single Spelling: Unambiguous
  - Multiple Spellings: Homophone IDs, Lexical misspellings
- Multiple Lexical Categories
  - Non-reserved keywords: Ambiguous interpretations
  - Homophones
## Input Stream Classification

<table>
<thead>
<tr>
<th>Single Spelling</th>
<th>Multiple Spellings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unambiguous</td>
<td>Homophone IDs</td>
</tr>
<tr>
<td></td>
<td>Lexical misspellings</td>
</tr>
<tr>
<td>Non-reserved keywords</td>
<td>Ambiguous interpretations</td>
</tr>
<tr>
<td></td>
<td>Homophones</td>
</tr>
</tbody>
</table>

Embedded Languages Fall in all Four Categories!
GLR Analysis Architecture

```c
for (i = 0; i < 10; i++) {
    I
}
```
GLR Analysis Architecture

for (i = 0; i < 10; i++) {
    I
}

Handles syntactic ambiguities

Lexer -- GLR Parser -- Semantics

FOR ( I

April 3, 2004  LDTA 2004
Our Contribution: XGLR Analysis Architecture

for i equals zero ...

Lexer -> XGLR Parser -> Semantics

FOR  I

FOR I
Our Contribution: XGLR Analysis Architecture

for i equals zero ...

Handles input stream ambiguities

Lexer → XGLR Parser → Semantics

FOR 4 I EYE

FOR I
LR Parsing

Parse Stack

1

Input Stream

FOR KW I ID = KW 0 #

Parse Table

<table>
<thead>
<tr>
<th></th>
<th>ID</th>
<th>KW</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>Err</td>
</tr>
<tr>
<td>2</td>
<td>R1</td>
<td>S4</td>
<td>Err</td>
</tr>
<tr>
<td>3</td>
<td>S9</td>
<td>R3</td>
<td>S7</td>
</tr>
</tbody>
</table>
LR Parsing

Parse Stack

1

Input Stream

FOR  I  =  0

KW  ID  KW  #

Parse Table

<table>
<thead>
<tr>
<th></th>
<th>ID</th>
<th>KW</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>Err</td>
</tr>
<tr>
<td>2</td>
<td>R1</td>
<td>S4</td>
<td>Err</td>
</tr>
<tr>
<td>3</td>
<td>S9</td>
<td>R3</td>
<td>S7</td>
</tr>
</tbody>
</table>
LR Parsing

Parse Stack

Input Stream

Parse Table

<table>
<thead>
<tr>
<th></th>
<th>ID</th>
<th>KW</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>Err</td>
</tr>
<tr>
<td>2</td>
<td>R1</td>
<td>S4</td>
<td>Err</td>
</tr>
<tr>
<td>3</td>
<td>S9</td>
<td>R3</td>
<td>S7</td>
</tr>
</tbody>
</table>
GLR Parsing

### Parse Stack

1

### Input Stream

<table>
<thead>
<tr>
<th>FOR</th>
<th>I</th>
<th>=</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>KW</td>
<td>ID</td>
<td>KW</td>
<td>#</td>
</tr>
</tbody>
</table>

### Parse Table

<table>
<thead>
<tr>
<th></th>
<th>ID</th>
<th>KW</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>Err</td>
</tr>
<tr>
<td></td>
<td>R5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>R1</td>
<td>S4</td>
<td>Err</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S9</td>
<td>R3</td>
<td>S7</td>
</tr>
</tbody>
</table>
GLR Parsing

**Parse Stack**

**Input Stream**
- FOR KW = KW
- I ID = KW
- 0 #

**Parse Table**

<table>
<thead>
<tr>
<th></th>
<th>ID</th>
<th>KW</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td>Err</td>
</tr>
<tr>
<td>2</td>
<td>R1</td>
<td>S4</td>
<td>Err</td>
</tr>
<tr>
<td>3</td>
<td>S9</td>
<td>R3</td>
<td>S7</td>
</tr>
</tbody>
</table>
GLR Parsing

Parse Stack

Input Stream

FOR KW I ID = KW 0 #

Parse Table

<table>
<thead>
<tr>
<th></th>
<th>ID</th>
<th>KW</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S2</td>
<td>S3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R5</td>
<td>Err</td>
</tr>
<tr>
<td>2</td>
<td>R1</td>
<td>S4</td>
<td>Err</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>S9</td>
<td>R3</td>
<td>S7</td>
</tr>
</tbody>
</table>
GLR Parsing

Parse Stack

Input Stream

Parse Table

Input Stream:

Parse Table:

<table>
<thead>
<tr>
<th>ID</th>
<th>KW</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S2</td>
<td>S3 R5</td>
</tr>
<tr>
<td>2</td>
<td>R1</td>
<td>S4</td>
</tr>
<tr>
<td>3</td>
<td>S9</td>
<td>R3</td>
</tr>
</tbody>
</table>
XGLR in Action

- Single Lexical Category
  - Single Spelling: Not Shown
  - Multiple Spellings: Example 1

- Multiple Lexical Categories
  - Single Spelling: Example 2
  - Multiple Spellings: Example 1
Parsing Homophones
XGLR Extension: Multiple Spellings, Single and Multiple Lexical Categories

FOUR
FORE
ID
BAR
KW
4
NUM

23
XGLR Extension: Parsers fork due to input ambiguity
Each parser shifts its now unambiguous input
The next input is lexed unambiguously
ID is only a valid lookahead for two parsers
Parsing Embedded Languages

Example BNF Grammar
Contains Languages L and W

L

\[ b_L \rightarrow \text{loop}_L \ d_W \ \text{END}_L \]
\[ \text{loop}_L \rightarrow \text{LOOP}_{L} \ | \ \varepsilon \]
\[ d_W \rightarrow \text{WHILE}_W \ \text{NUM}_W \ \text{do}_W \]
\[ \text{do}_W \rightarrow \text{DO}_W \ | \ \varepsilon \]
Parsing Embedded Languages

Example BNF Grammar
Contains Languages L and W

\[ L \]

\[ b_L \rightarrow \text{loop}_L \ d_W \ \text{END}_L \]
\[ \text{loop}_L \rightarrow \text{LOOP}_L \ | \ \varepsilon \]
\[ d_W \rightarrow \text{WHILE}_W \ \text{NUM}_W \ \text{do}_W \]
\[ \text{do}_W \rightarrow \text{DO}_W \ | \ \varepsilon \]

\[ \text{LOOP \ WHILE \ 34 \ END} \]
\[ \text{WHILE \ 56 \ DO \ END} \]
State 0
S' → • b_L, $

b_L → • LOOP_L d_w END_L, $

loop_L → •, WHILE_w

loop_L → • LOOP_L, WHILE_w

State 1
b_L → LOOP_L • d_w END_L, $

d_w → • WHILE_w NUM_w do_w, END_L

State 2
d_w → WHILE_w • NUM_w do_w, END_L

State 3
d_w → WHILE_w NUM_w • DO_w, END_L
do_w → •, END_L
do_w → • DO_w, END_L

State 4
loop_L → LOOP_L •, WHILE_w

State 5
b_L → LOOP_L d_w • END_L, $

State 6
b_L → LOOP_L d_w END_L •, $

State 7
b_L → LOOP_L d_w END_L •, END_L

d_w → WHILE_w NUM_w DO_w •, END_L

State 8
do_w → DO_w •, END_L
Parsing Embedded Languages

State 0
- $S' \rightarrow b_L \cdot \$, $b_L \rightarrow \text{loop}_L \cdot d_w \text{END}_L \cdot \$
- $\text{loop}_L \rightarrow \cdot \, \text{WHILE}_w$
- $\text{loop}_L \rightarrow \cdot \text{LOOP}_L \cdot \text{WHILE}_w$

State 4
- $\text{loop}_L \rightarrow \text{LOOP}_L \cdot \, \text{WHILE}_w$

State 1
- $b_L \rightarrow \text{loop}_L \cdot d_w \text{END}_L \cdot \$
- $d_w \rightarrow \cdot \text{WHILE}_w \text{NUM}_w \text{do}_w \cdot \text{END}_L$
Current parse state has ambiguous lexical language
XGLR Extension: Fork parsers, assign one to each lexical language
XGLR Extension: Single spelling, Multiple lexical categories
Lex lookahead both in language L and W
Only \( \text{LOOP}_L \) is valid lookahead, and is shifted
XGLR Extension: State 4 has lexer lookaheads only in language W
Lex lookahead in language W

State 0
S' → • b_L, $

b_L → • loop_L d_w END_L, $

loop_L → • , WHILE_w

loop_L → • LOOP_L, WHILE_w

State 1
b_L → loop_L d_w END_L, $

d_w → • WHILE_w NUM_w d_o_w, END_L

State 4
loop_L → LOOP_L •, WHILE_w
REDUCE by rule 2 and GOTO state 1
Shift into state 2

State 1
\[ b_L \rightarrow \text{loop}_L \cdot d_w \cdot \text{END}_L \cdot \$, \\
\[ d_w \rightarrow \cdot \text{WHILE}_w \cdot \text{NUM}_w \cdot \text{DO}_w \cdot \text{END}_L \]

State 2
\[ d_w \rightarrow \text{WHILE}_w \cdot \text{NUM}_w \cdot \text{DO}_w \cdot \text{END}_L \]

State 3
\[ d_w \rightarrow \text{WHILE}_w \cdot \text{NUM}_w \cdot \text{DO}_w \cdot \text{END}_L \]
XGLR Extension: Lex lookahead in language W

State 1
\[ b_L \rightarrow \text{loop}_L \cdot d_w \text{ END}_L, \$ \]
\[ d_w \rightarrow \bullet \text{ WHILE}_w \text{ NUM}_w \text{ DO}_w, \text{ END}_L \]

State 2
\[ d_w \rightarrow \text{ WHILE}_w \cdot \text{ NUM}_w \text{ do}_w, \text{ END}_L \]

State 3
\[ d_w \rightarrow \text{ WHILE}_w \text{ NUM}_w \cdot \text{ do}_w, \text{ END}_L \]
\[ d_o \rightarrow \bullet, \text{ END}_L \]
\[ d_o \rightarrow \bullet \text{ DO}_w, \text{ END}_L \]
State 1
\( b_L \rightarrow \text{loop}_L \cdot d_w \cdot \text{END}_L \cdot \$ \)
\( d_w \rightarrow \cdot \text{WHILE}_w \cdot \text{NUM}_w \cdot \text{DO}_w \cdot \text{END}_L \)

State 2
\( d_w \rightarrow \text{WHILE}_w \cdot \text{NUM}_w \cdot d_w \cdot \text{END}_L \)

State 3
\( d_w \rightarrow \text{WHILE}_w \cdot \text{NUM}_w \cdot d_w \cdot \text{END}_L \)
\( d_w \rightarrow \cdot \), \text{END}_L \)
\( d_w \rightarrow \cdot \text{DO}_w \cdot \text{END}_L \)
Shift into state 3

State 1
\[ b_L \rightarrow \text{loop}_L \cdot d_w \cdot \text{END}_L \cdot \$, \\
\[ d_w \rightarrow \cdot \text{WHILE}_w \cdot \text{NUM}_w \cdot \text{DO}_w \cdot \text{END}_L \]

State 2
\[ d_w \rightarrow \text{WHILE}_w \cdot \text{NUM}_w \cdot d_o \cdot \text{END}_L \]

State 3
\[ d_w \rightarrow \text{WHILE}_w \cdot \text{NUM}_w \cdot d_o \cdot \text{END}_L \]

\[ d_o \rightarrow \cdot \cdot \cdot \; \text{END}_L \]

\[ d_o \rightarrow \cdot \; \text{DO}_w \cdot \text{END}_L \]
Shift into state 3, which has ambiguous lexical language
XGLR Extension: Single spelling, Multiple lexical categories
Fork parsers, assign one to each lexical language

State 1
\[ b_L \rightarrow \text{loop}_L \cdot d_w \cdot \text{END}_L \cdot \$ \]
\[ d_w \rightarrow \bullet \cdot \text{WHILE}_w \cdot \text{NUM}_w \cdot \text{DO}_w \cdot \text{END}_L \]

State 2
\[ d_w \rightarrow \text{WHILE}_w \cdot \text{NUM}_w \cdot \text{DO}_w \cdot \text{END}_L \]

State 3
\[ d_w \rightarrow \text{WHILE}_w \cdot \text{NUM}_w \cdot \text{DO}_w \cdot \text{END}_L \]
\[ d_w \rightarrow \bullet \cdot \text{END}_L \]
GLR Ambiguity Support

1. Fork parser on shift-reduce conflict
2. Fork parser on reduce-reduce conflict
XGLR Ambiguity Support

1. Fork parser on shift-reduce conflict
2. Fork parser on reduce-reduce conflict
XGLR Ambiguity Support

1. Fork parser on shift-reduce conflict
2. Fork parser on reduce-reduce conflict
3. Fork parsers on ambiguous lexical language
   - Single spelling, Multiple lexical categories
4. Fork parsers on ambiguous lexical lookahead
   - Single/Multiple Spellings, Multiple lexical categories
   - Shift-shift conflict resolution
XGLR Ambiguities

- Many GLR programming language specs have finite, few ambiguities
- XGLR language specs *also* have finite, but slightly more, ambiguities
  - Lexical ambiguity due to ambiguous input does result in more ambiguous parse forests
XGLR Ambiguities

- Many GLR programming language specs have finite, few ambiguities
- XGLR language specs also have finite, but slightly more, ambiguities
  - Lexical ambiguity due to ambiguous input does result in more ambiguous parse forests
- Ambiguity causes parsers to fork
- GLR maintains efficiency by merging parsers when ambiguity is over
Parser Merging

- GLR: Parsers merge when in same parse state
Parser Merging

- GLR: Parsers merge when in same parse state
Parser Merging

- XGLR: Parsers merge when in same parse state and same lexical state
Parser Merging

- XGLR: Parsers merge when in same parse state and same lexical state
Parser Merging

- XGLR: Parsers merge when in same parse state and same lexical state
Parser Merging

- XGLR: Parsers merge when in same parse state \textit{and} same lexical state
Parser Merging

- XGLR: Parsers merge when in same parse state and same lexical state
Out of Sync Parsers

- XGLR: Parsers merge when in same parse state and same lexical state and same input position

DO57I=3
Out of Sync Parsers

- XGLR: Parsers merge when in the same parse state and the same lexical state and same input position.

Diagram:

```
   8
  / 
A   W
  
5  A
  
1  A
      
DO

W

57I=3

ID

DO57I

W

A

KW

=3
```
Out of Sync Parsers

- XGLR: Parsers merge when in same parse state and same lexical state and same input position
Out of Sync Parsers

- XGLR: Parsers merge when in same parse state and same lexical state and same input position
Out of Sync Parsers

- XGLR: Parsers merge when in same parse state and same lexical state and same input position.
Out of Sync Parsers

- XGLR: Parsers merge when in same parse state and same lexical state and same input position

Diagram:

1. DO
2. 57
3. 5
4. =
5. 6
6. 3
7. A
8. W
Out of Sync Parsers

- XGLR: Parsers merge when in same parse state and same lexical state and same input position
Out of Sync Parsers

- XGLR: Parsers merge when in same parse state and same lexical state and same input position.
Implementation

- Keep map: lookahead $\rightarrow$ parser to use when looking for parsers to merge with

- Sort parsers by position of lookahead in the input
  - Enables pruning of map as parsers move past a particular input location
  - Extra memory required is bounded by dynamic separation between first and last parsers
Related Work

- **GLR Parsing Algorithm**
  - Tomita [1985]
  - Farshi [1991]
  - Rekers [1992]
  - Johnstone *et. al.* [2002]

- **Incremental GLR**
  - Wagner [1997]

- **GLR Implementations**
  (that I heard of before today)
  - ASF+SDF [1993]
  - Elkhound [2004]
  - Bison [2003]
  - DParser [2002]
  - Aycock and Horspool [1999]

- **Scannerless Parsing**
  (or Context-Free Scanning)
  - Salomon and Cormack [1989]
  - Visser [1997]
    - van den Brand [2002]

- **Ambiguous Input Streams**
  - Aycock and Horspool [2001]

- **Embedded Languages**
  - ASF+SDF [1997]
  - Van de Vanter and Boshernitsan (CodeProcessor) [2000]
Future Work

- Semantic Analysis of Embedded Languages
- Automated Semantic Disambiguation
Contributions

1. Generalized GLR to handle *input stream ambiguities*
2. Classified input stream ambiguities into four categories
3. Implemented XGLR algorithm in Harmonia framework
4. Constructed combined lexer and parser generator to support embedded languages and lexical ambiguities at each stage of analysis
5. Enabled analysis of embedded languages, programming by voice, and legacy languages