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

Human Speech

Harmonia: Language-aware Editing

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

Human Speech

Embedded Languages

Harmonia: Language-aware Editing

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

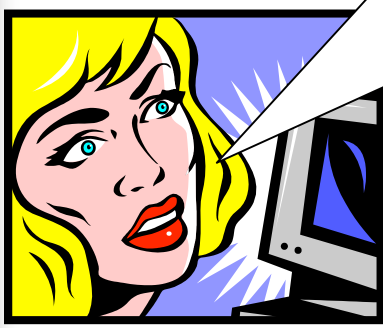
Human Speech

Embedded Languages

Each kind of input stream ambiguity requires new language analyses

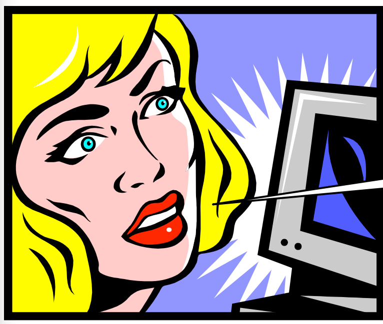
Speech Example

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



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

}
```

Ambiguities



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

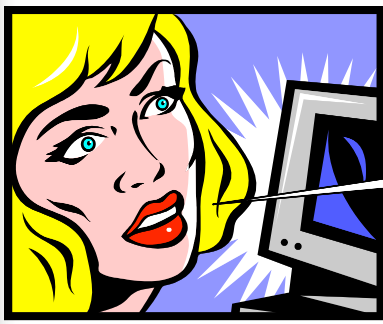
```
for (int i = 0; i < 10; i++ ) {
    ▌
}
```

Ambiguities

KW or #?

ID Spelling?

KW or ID?

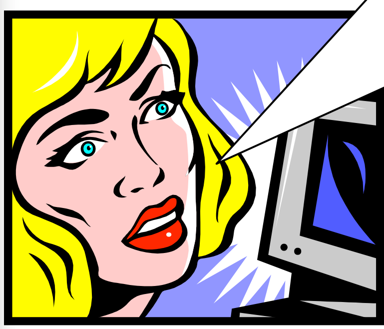


4 int *eye* equals *0* *aye* less *then* *10* i plus plus

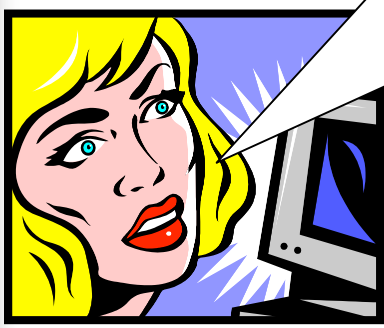
```
for (int i = 0; i < 10; i++ ) {

}
```

Another Utterance

for times ate equals zero two plus equals one



Many Valid Parses!




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

}
```

```
4 * 8 = zero; to += won
```

```
fore.times(8).equalsZero(2, plus == 1)
```

Embedded Language Example

- C and Regexps embedded in Flex

Flex Rule for Identifiers

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

Embedded Language Example

- C and Regexps embedded in Flex

<u>Flex Rule for Identifiers</u>

```
[_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

```
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

```
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
```

ID

ID

KW

ID

Input Stream Classification

	Single Spelling	Multiple Spellings
Single Lexical Category	Unambiguous	Homophone IDs Lexical misspellings
Multiple Lexical Categories	Non-reserved keywords Ambiguous interpretations	Homophones

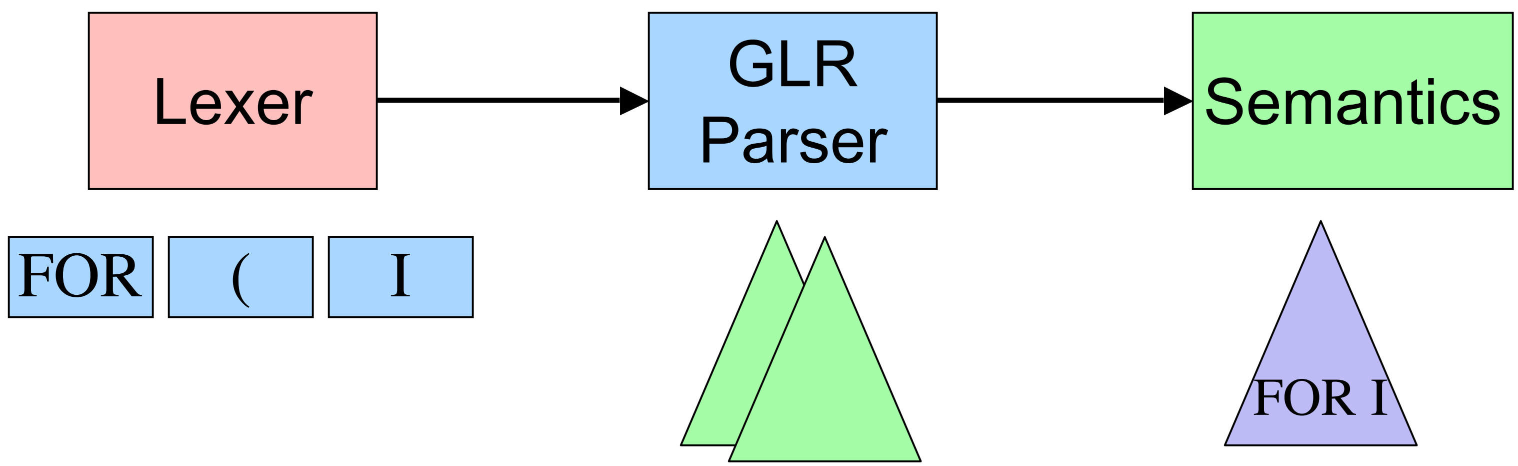
Input Stream Classification

	Single Spelling	Multiple Spellings
Single Lexical Category	Unambiguous	Homophone IDs Lexical misspellings
Multiple Lexical Categories	Non-reserved keywords Ambiguous interpretations	Homophones

Embedded Languages Fall in all Four Categories!

GLR Analysis Architecture

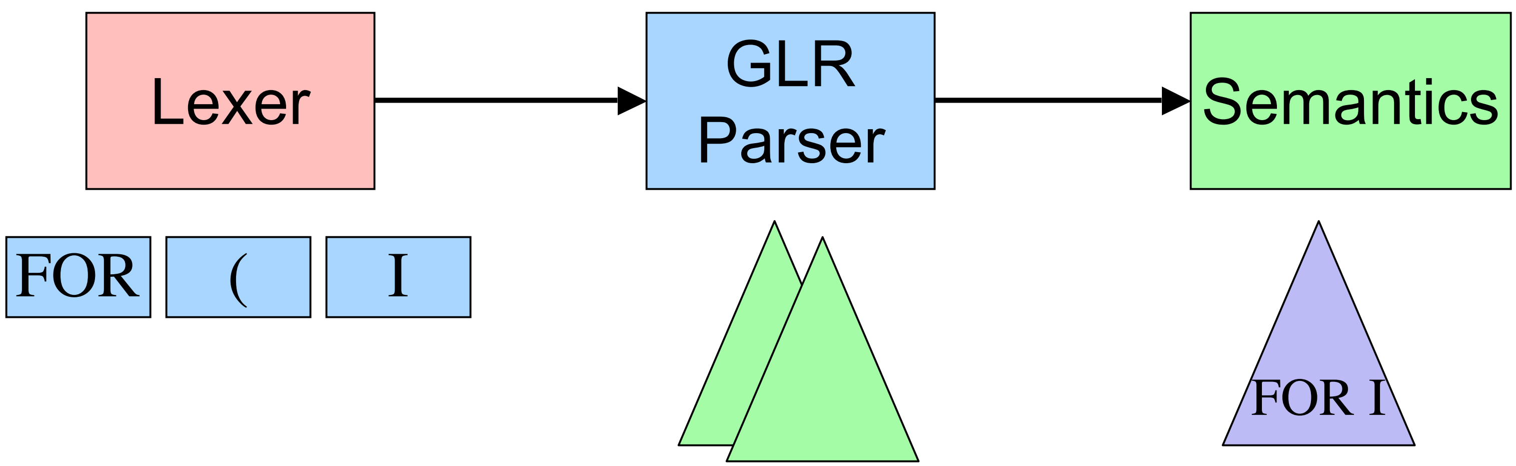
```
for (i = 0; i < 10; i++ ) {

}
```



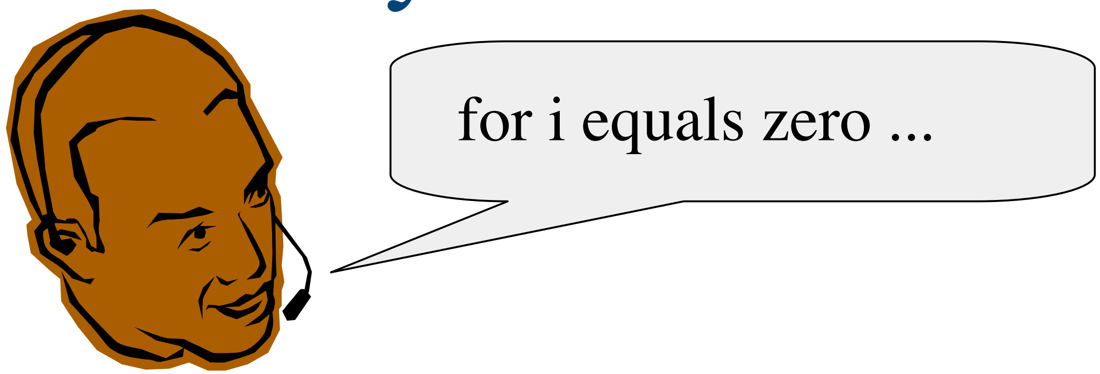
GLR Analysis Architecture

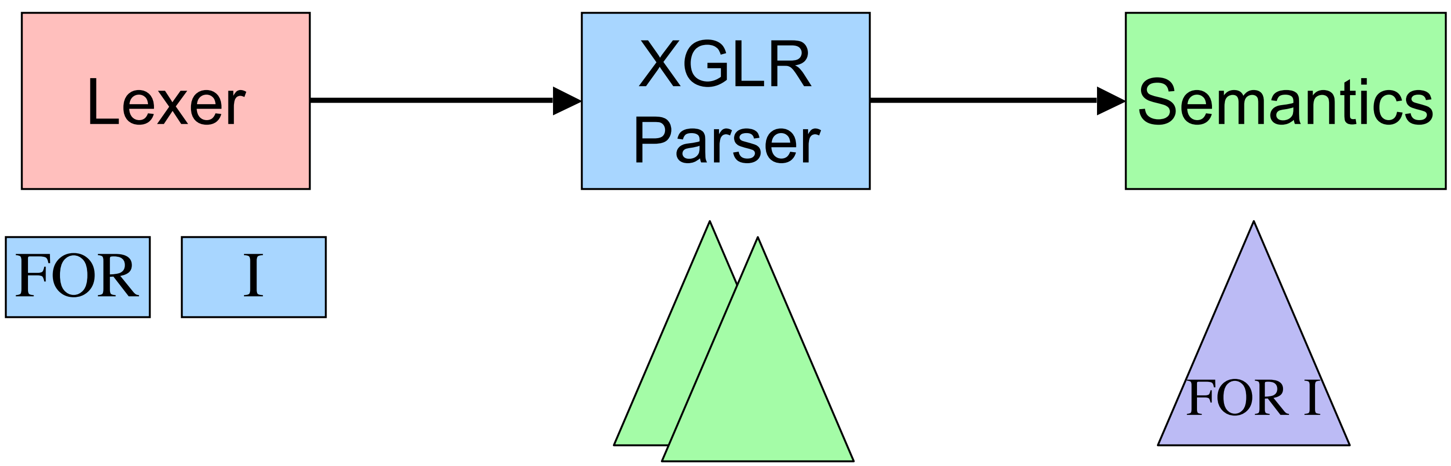
```
for (i = 0; i < 10; i++ ) {
   ▌
}
```

Handles syntactic ambiguities

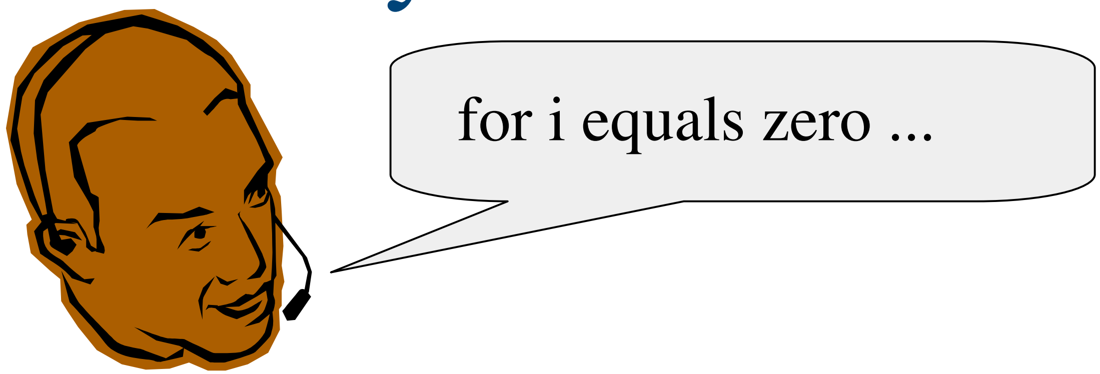


Our Contribution: XGLR Analysis Architecture

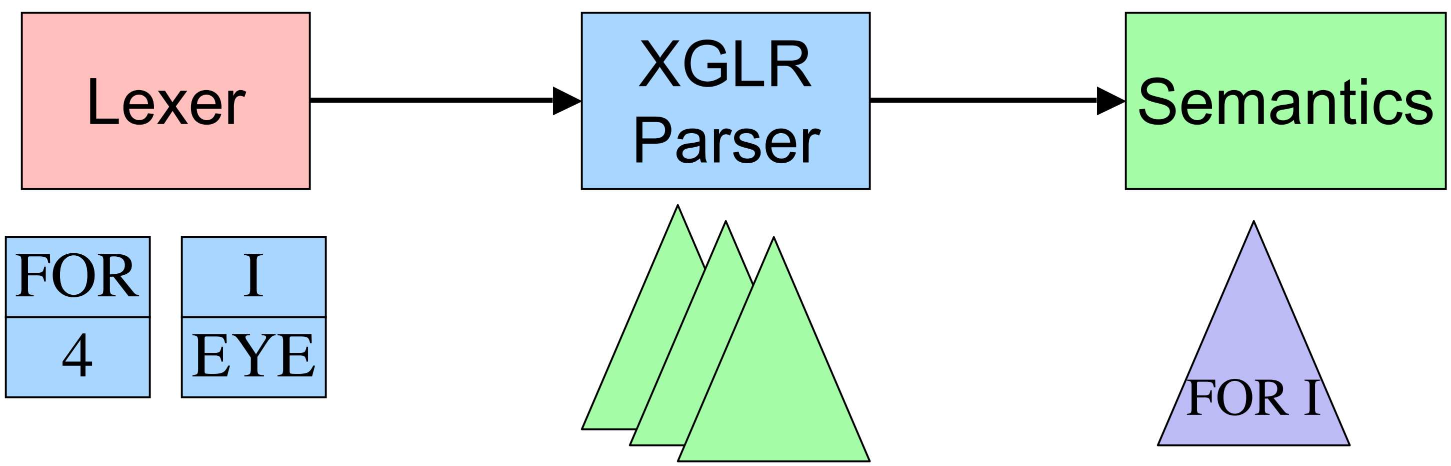




Our Contribution: XGLR Analysis Architecture



Handles *input stream* ambiguities



LR Parsing

Parse Stack

1

Input Stream

FOR (KW) | I (ID) | = (KW) | 0 (#)

Parse Table

	ID	KW	#
1	S2	S3	Err
2	R1	S4	Err
3	S9	R3	S7

LR Parsing

Parse Stack

1

Input Stream

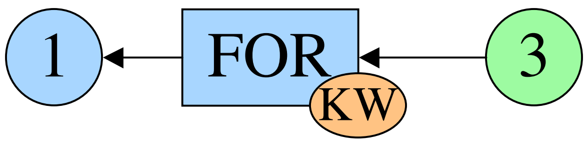
FOR KW | I ID | = KW | 0 #

Parse Table

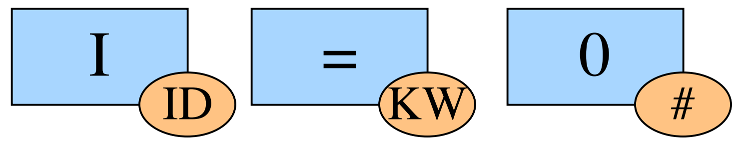
	ID	KW	#
1	S2	S3	Err
2	R1	S4	Err
3	S9	R3	S7

LR Parsing

Parse Stack



Input Stream



Parse Table

	ID	KW	#
1	S2	S3	Err
2	R1	S4	Err
3	S9	R3	S7

GLR Parsing

Parse Stack

1

Input Stream

FOR KW | I ID | = KW | 0 #

Parse Table

	ID	KW	#
1	S2	S3 R5	Err
2	R1 R2	S4	Err
3	S9	R3	S7

GLR Parsing

Parse Stack

1

Input Stream

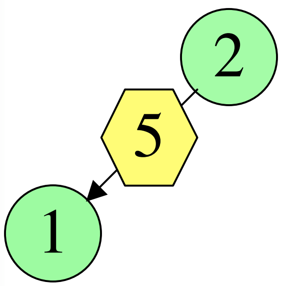
FOR KW | I ID | = KW | 0 #

Parse Table

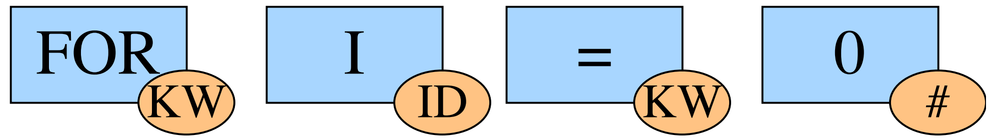
	ID	KW	#
1	S2	S3 R5	Err
2	R1 R2	S4	Err
3	S9	R3	S7

GLR Parsing

Parse Stack



Input Stream

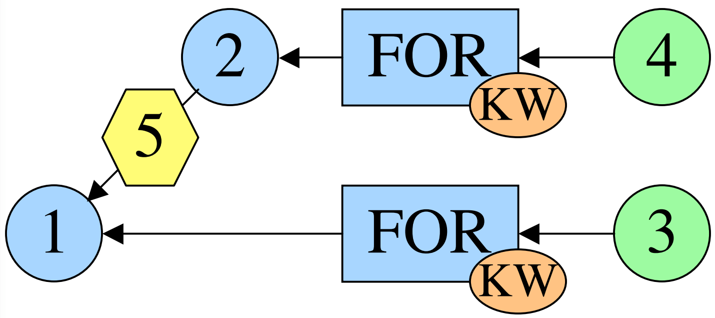


Parse Table

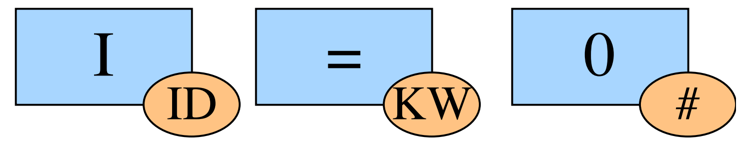
	ID	KW	#
1	S2	S3 R5	Err
2	R1 R2	S4	Err
3	S9	R3	S7

GLR Parsing

Parse Stack



Input Stream



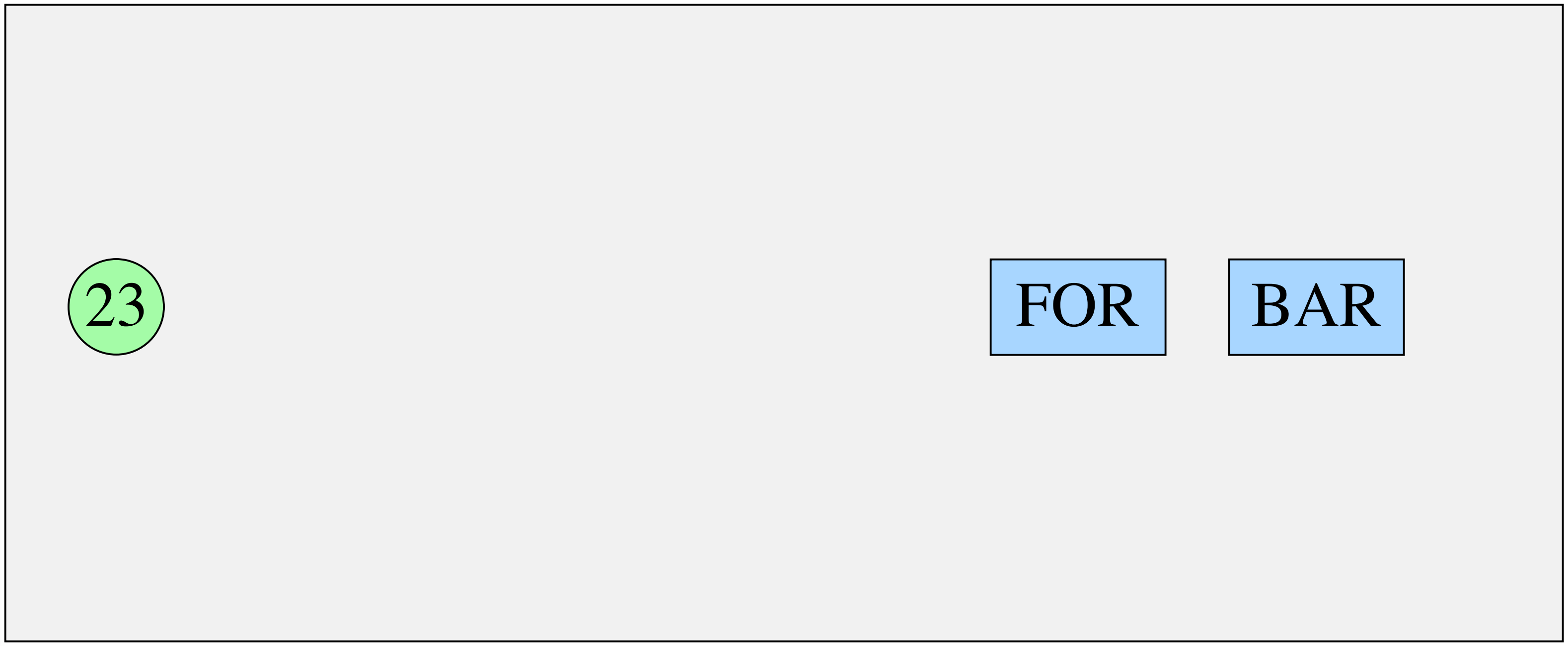
Parse Table

	ID	KW	#
1	S2	S3 R5	Err
2	R1 R2	S4	Err
3	S9	R3	S7

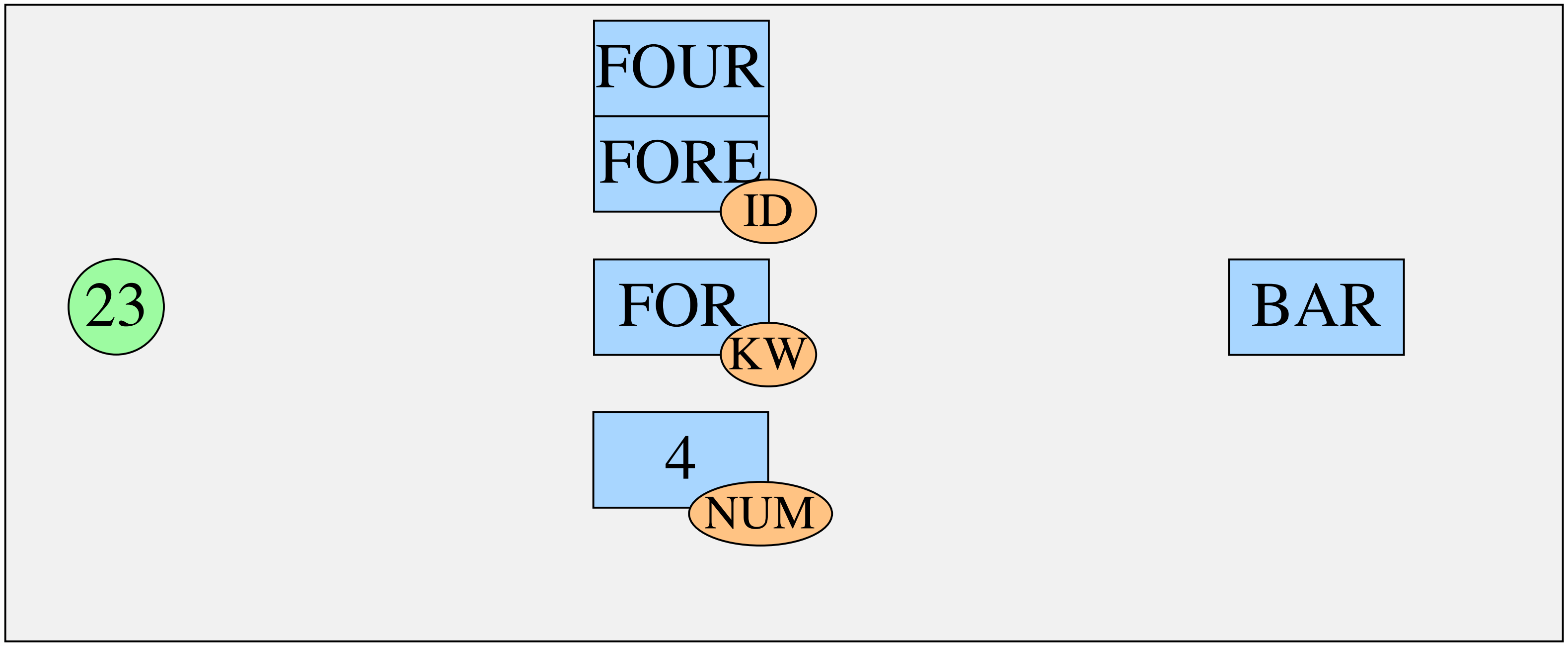
XGLR in Action

	Single Spelling	Multiple Spellings
Single Lexical Category	Not Shown	Example 1
Multiple Lexical Categories	Example 2	Example 1

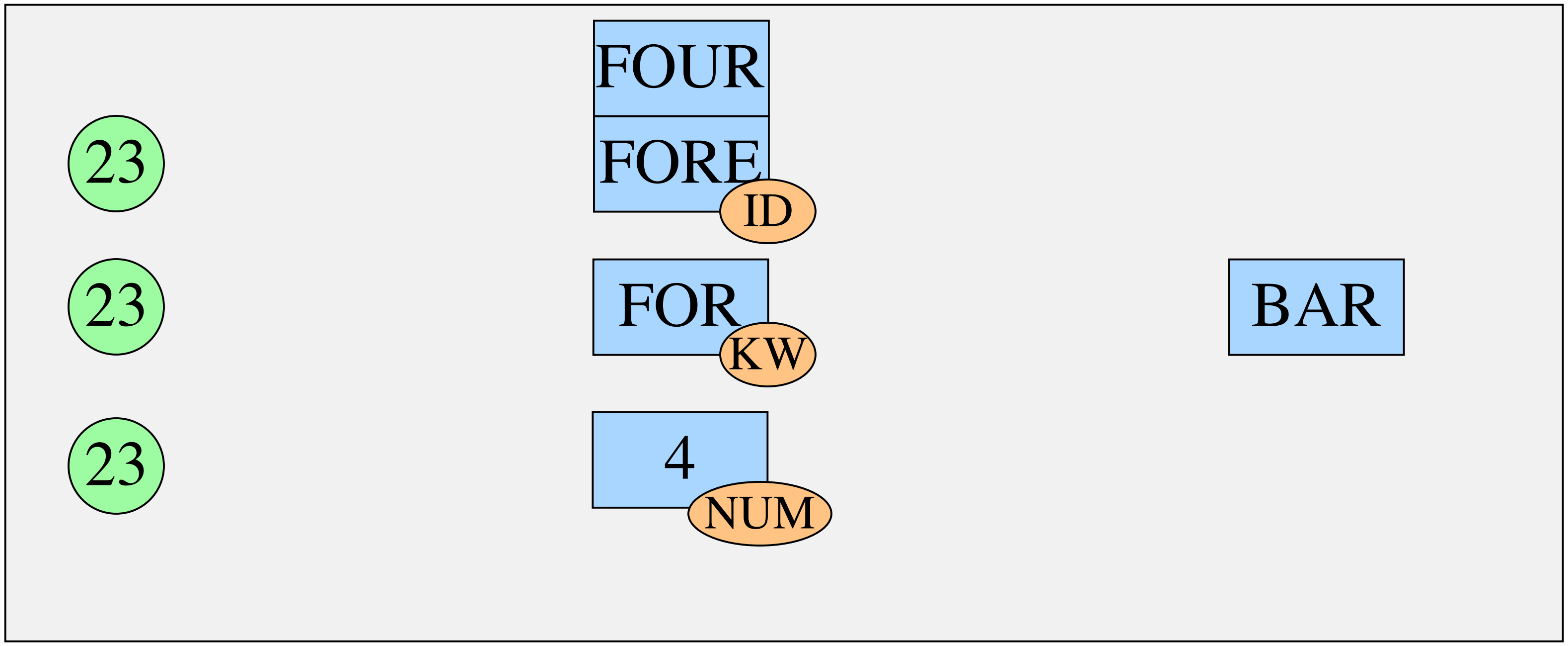
Parsing Homophones



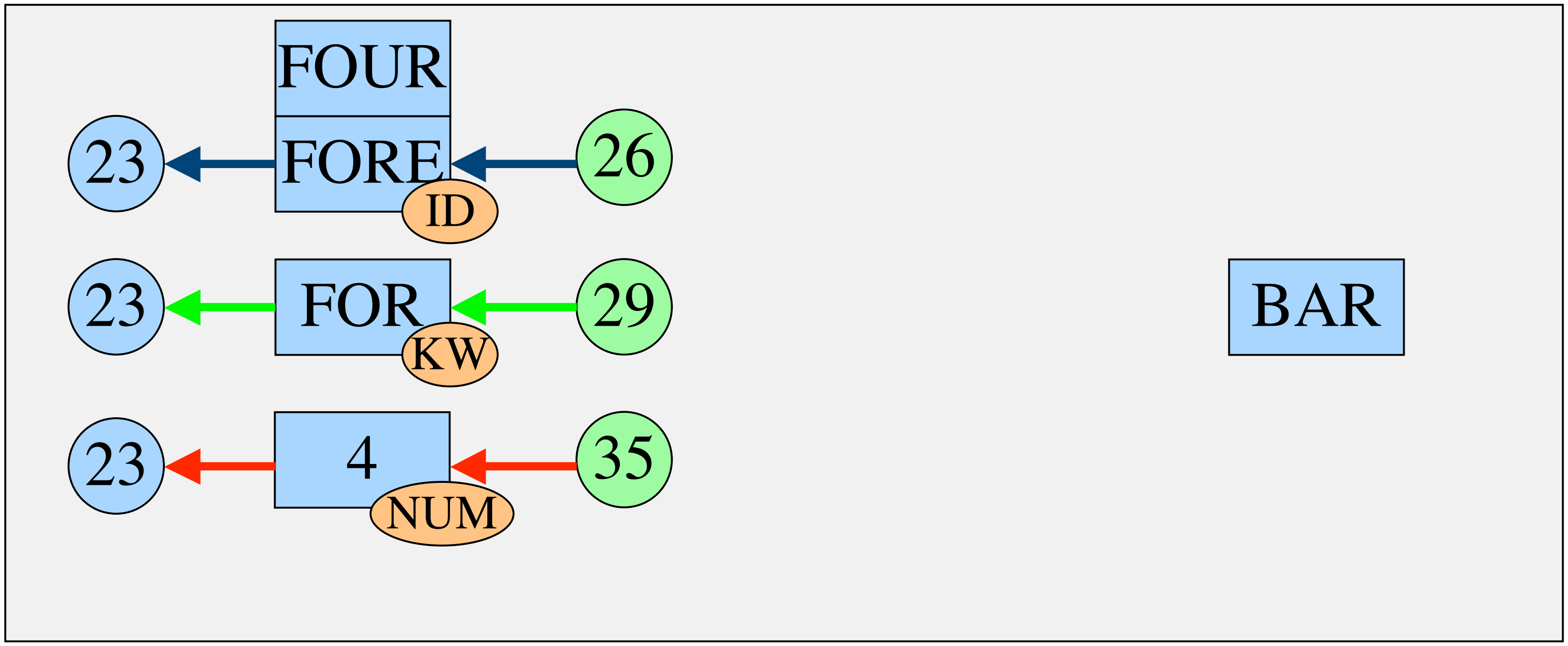
XGLR Extension: Multiple Spellings, Single and Multiple Lexical Categories



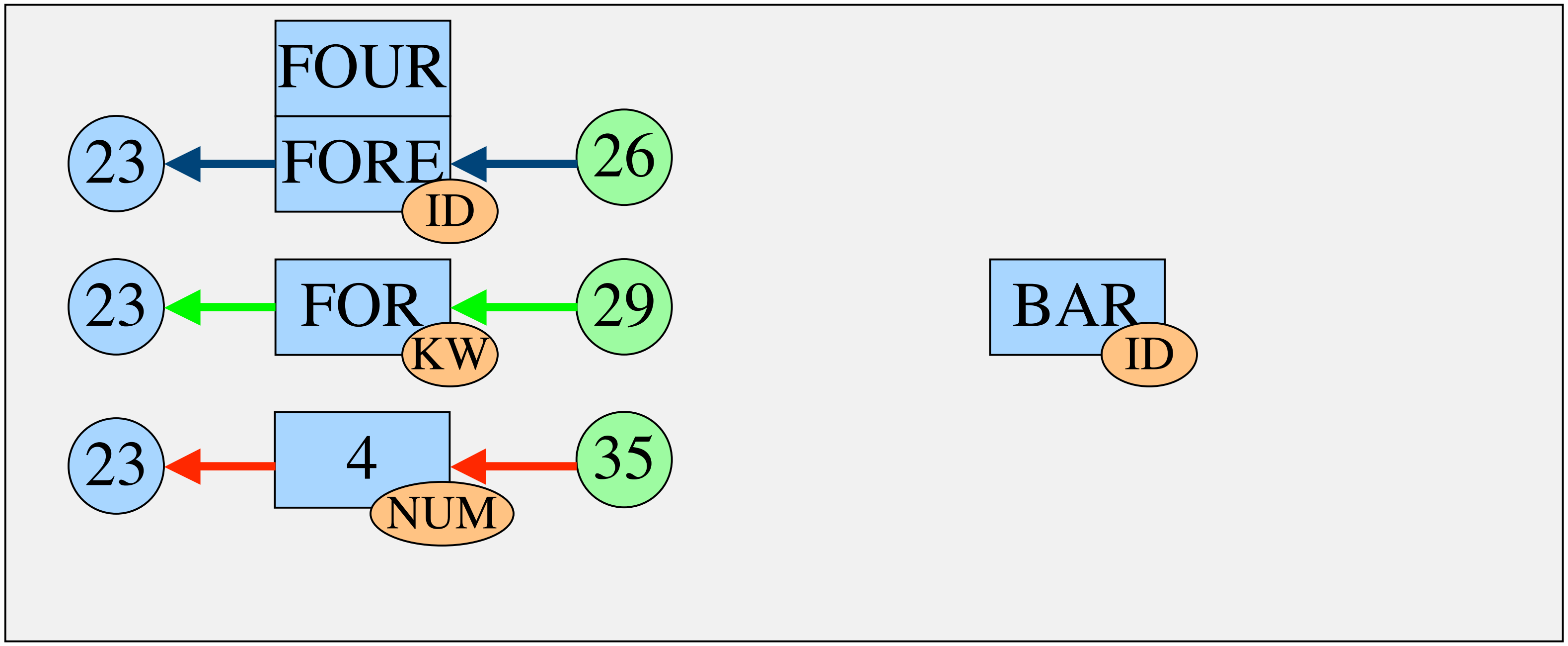
XGLR Extension: Parsers fork due to input ambiguity



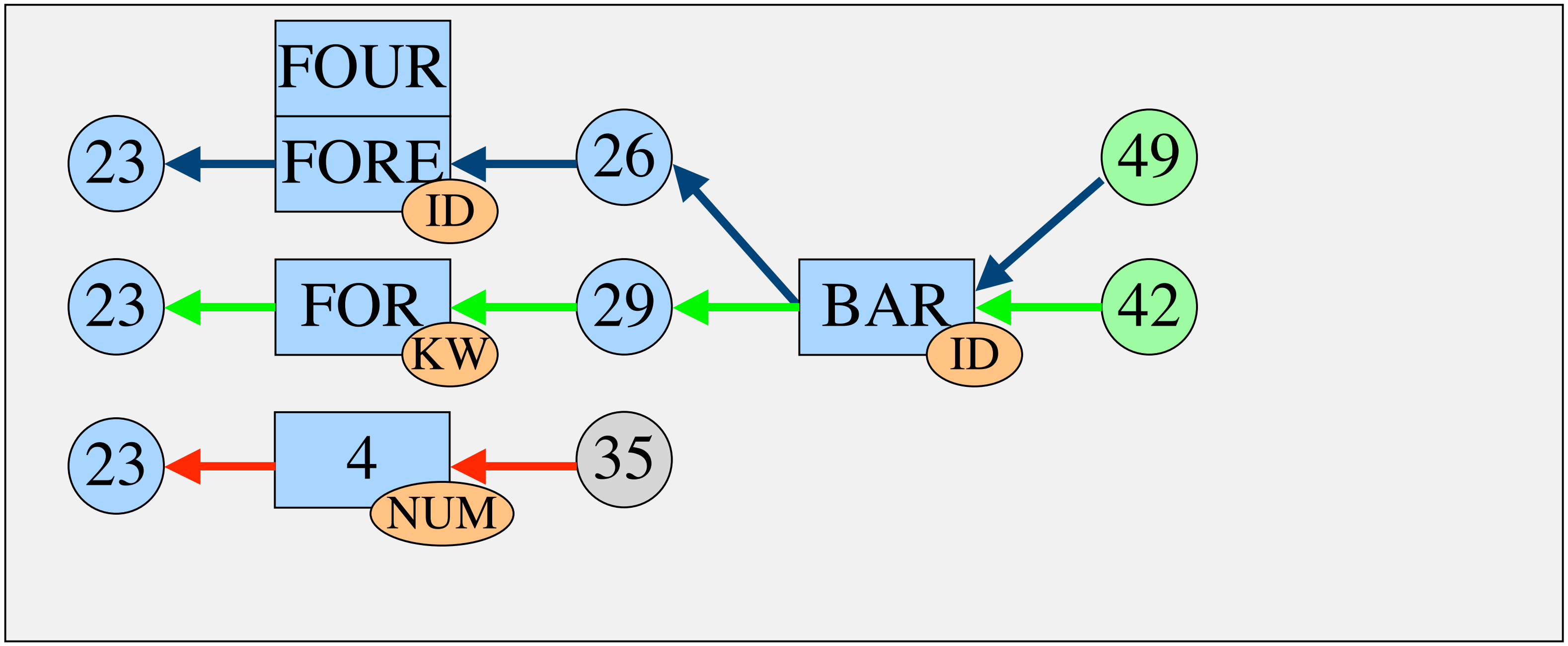
Each parser shifts its now unambiguous input



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

$b_L \rightarrow loop_L\ d_W\ END_L$

$loop_L \rightarrow LOOP_L \mid \varepsilon$

W

$d_W \rightarrow WHILE_W\ NUM_W\ do_W$

$do_W \rightarrow DO_W \mid \varepsilon$

Parsing Embedded Languages

Example BNF Grammar

Contains Languages L and W

L

$b_L \rightarrow loop_L\ d_W\ END_L$

$loop_L \rightarrow LOOP_L \mid \varepsilon$

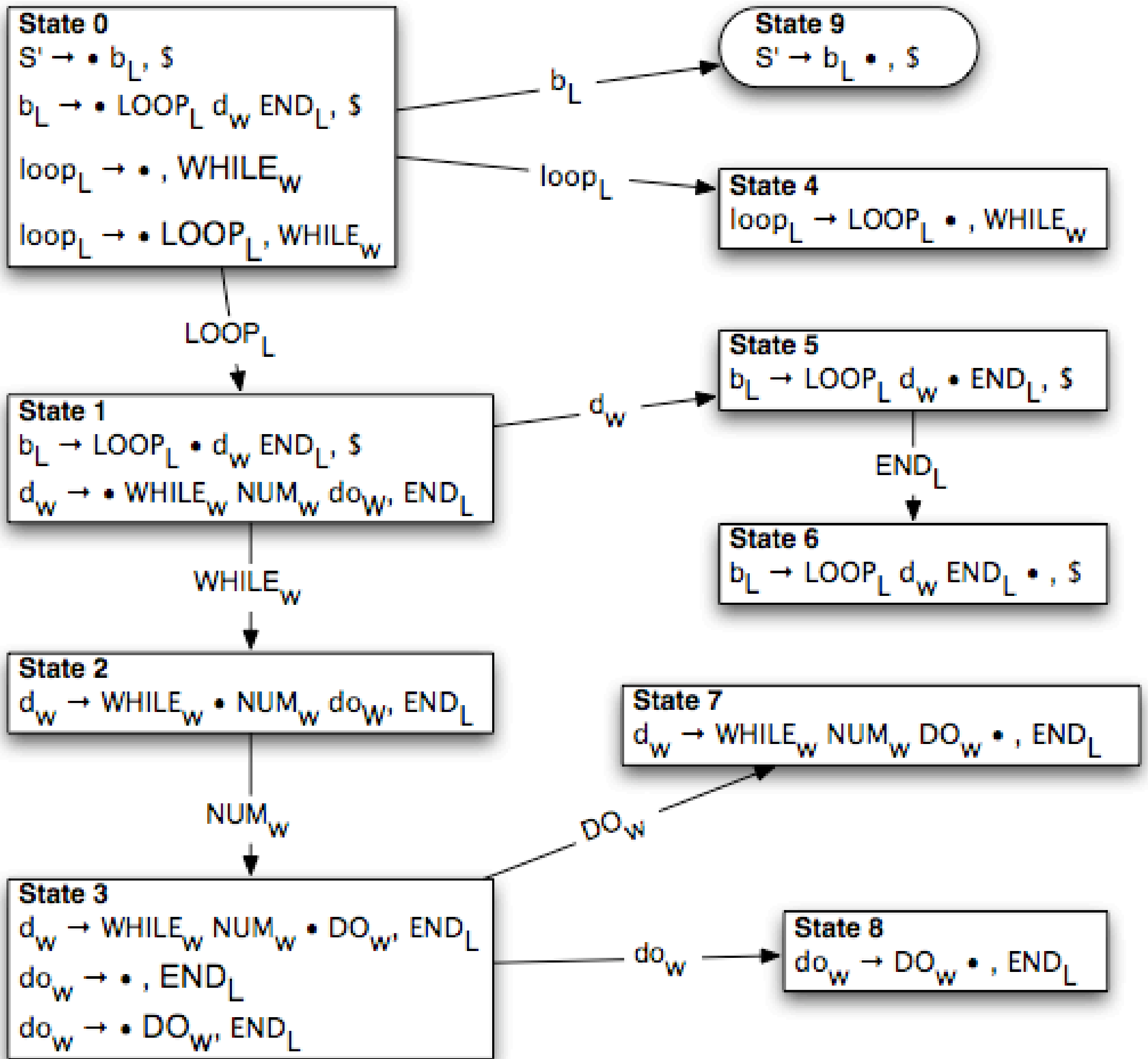
W

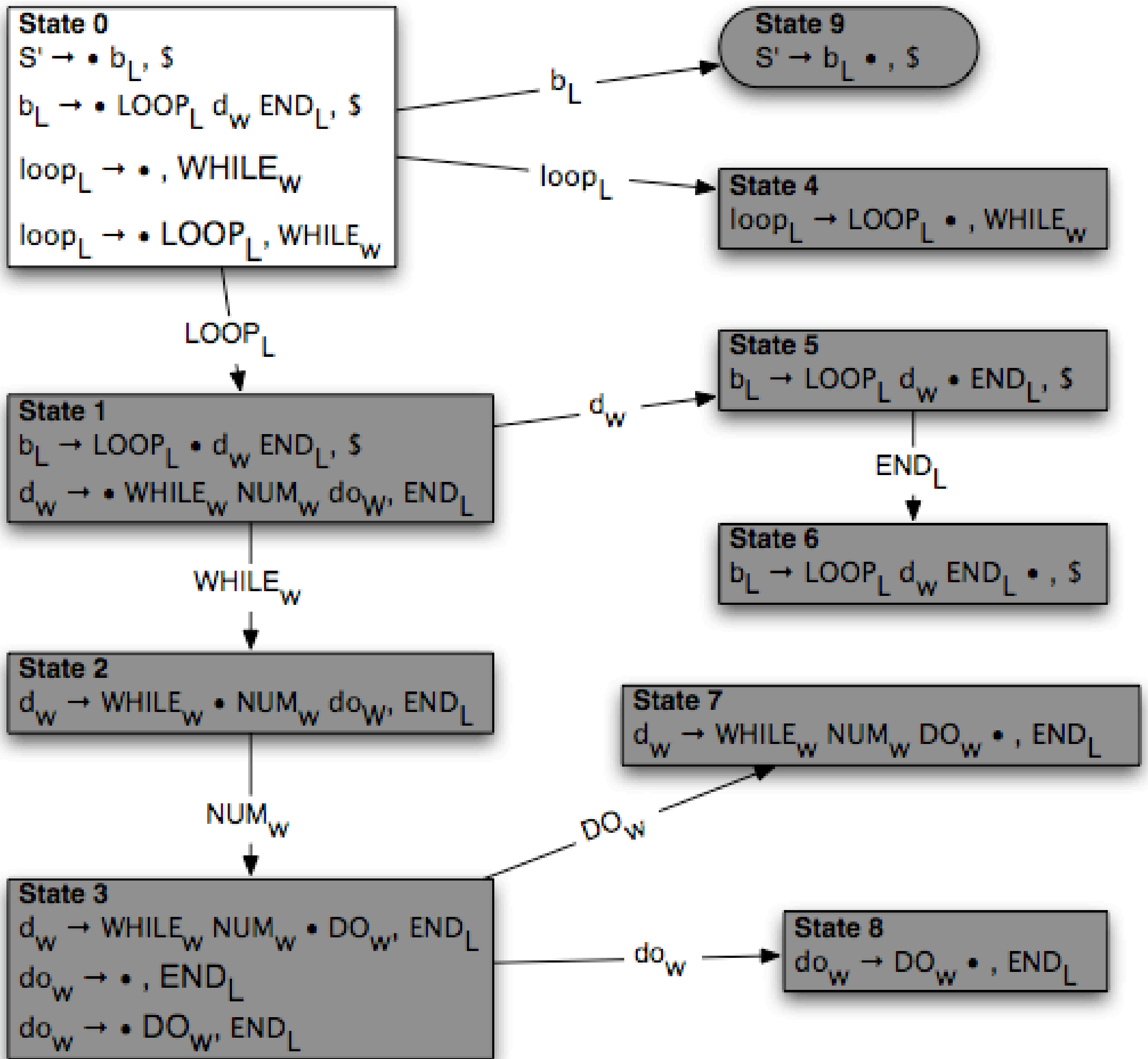
$d_W \rightarrow WHILE_W\ NUM_W\ do_W$

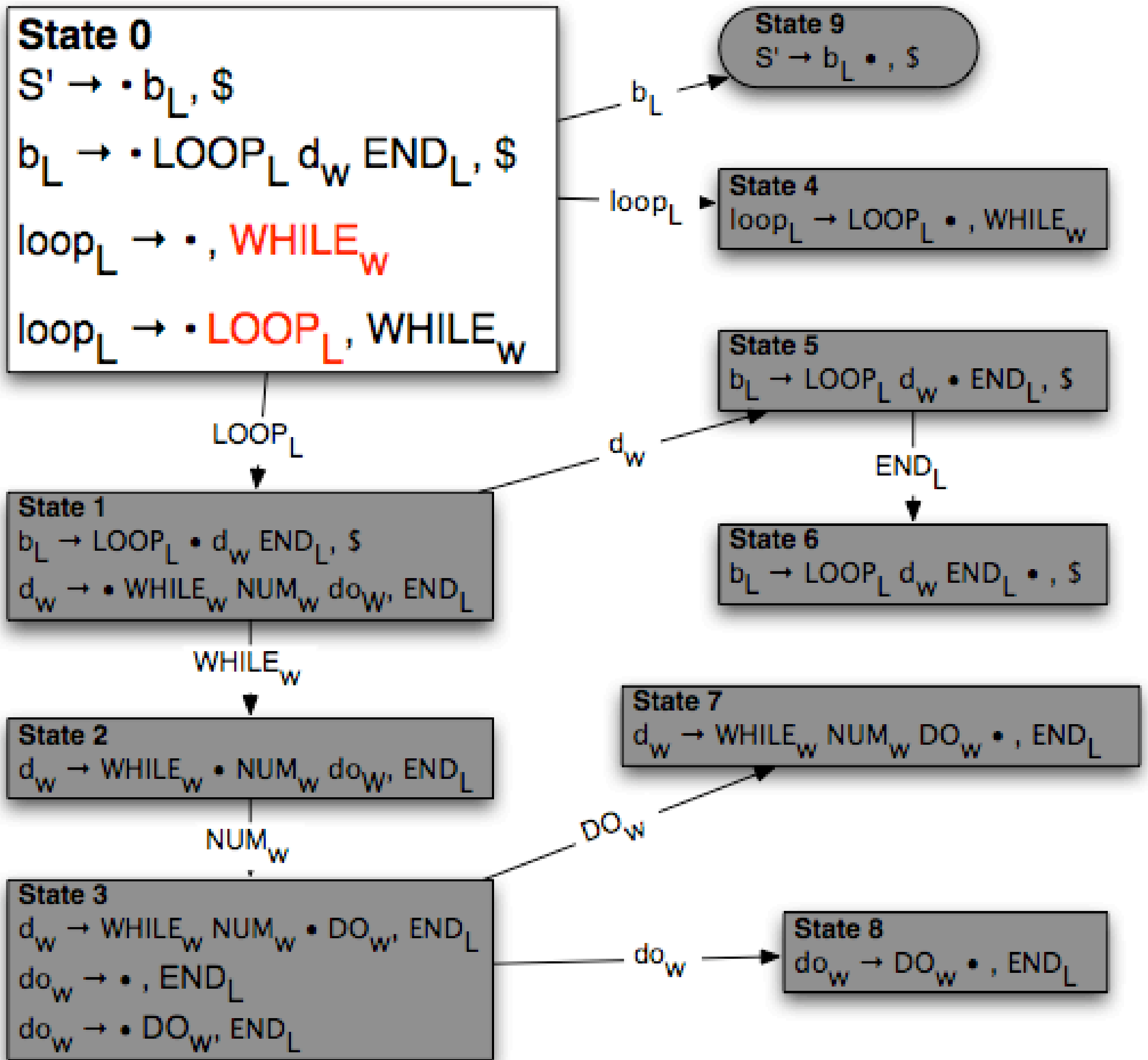
$do_W \rightarrow DO_W \mid \varepsilon$

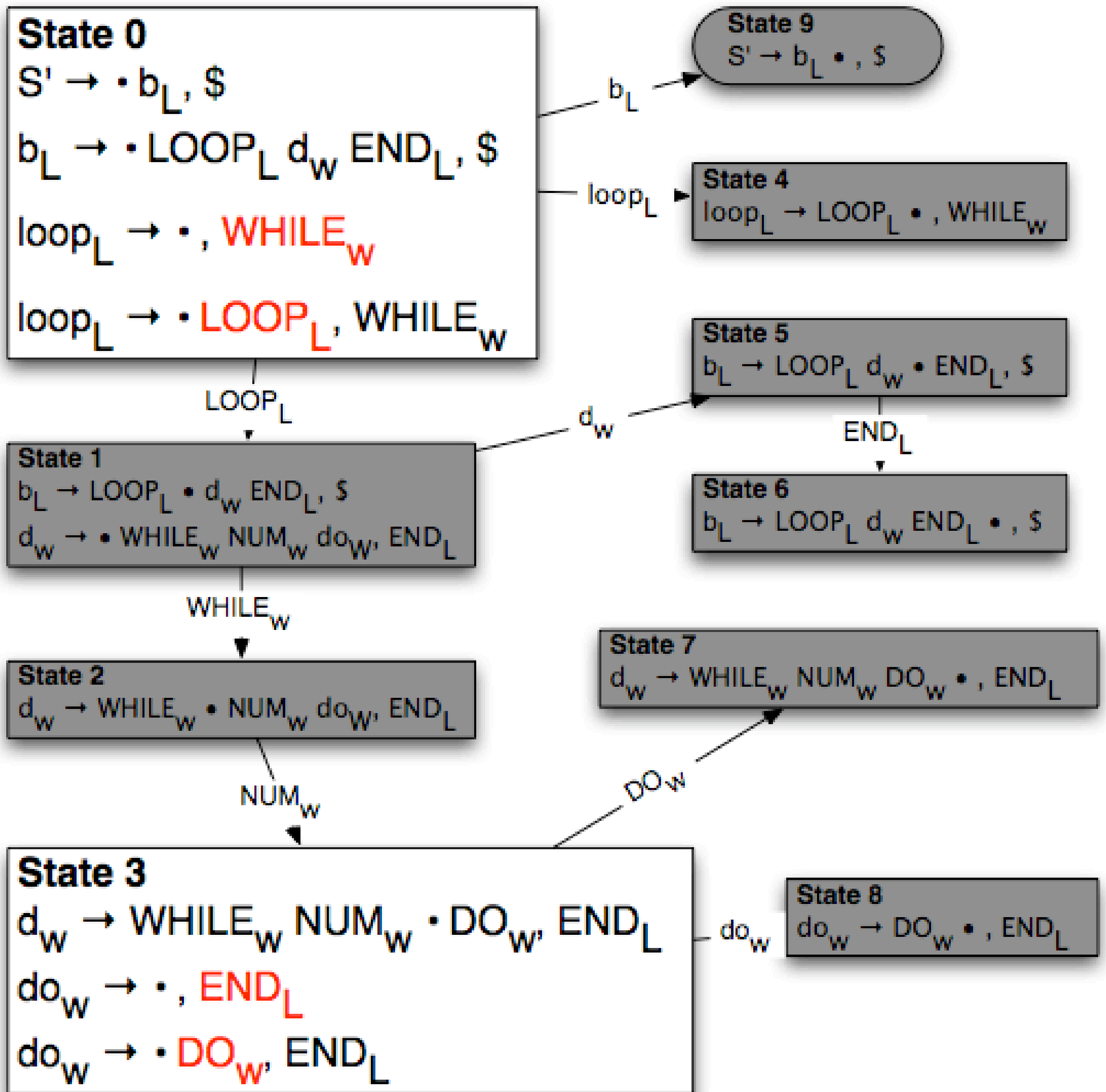
LOOP WHILE 34 END

WHILE 56 DO END

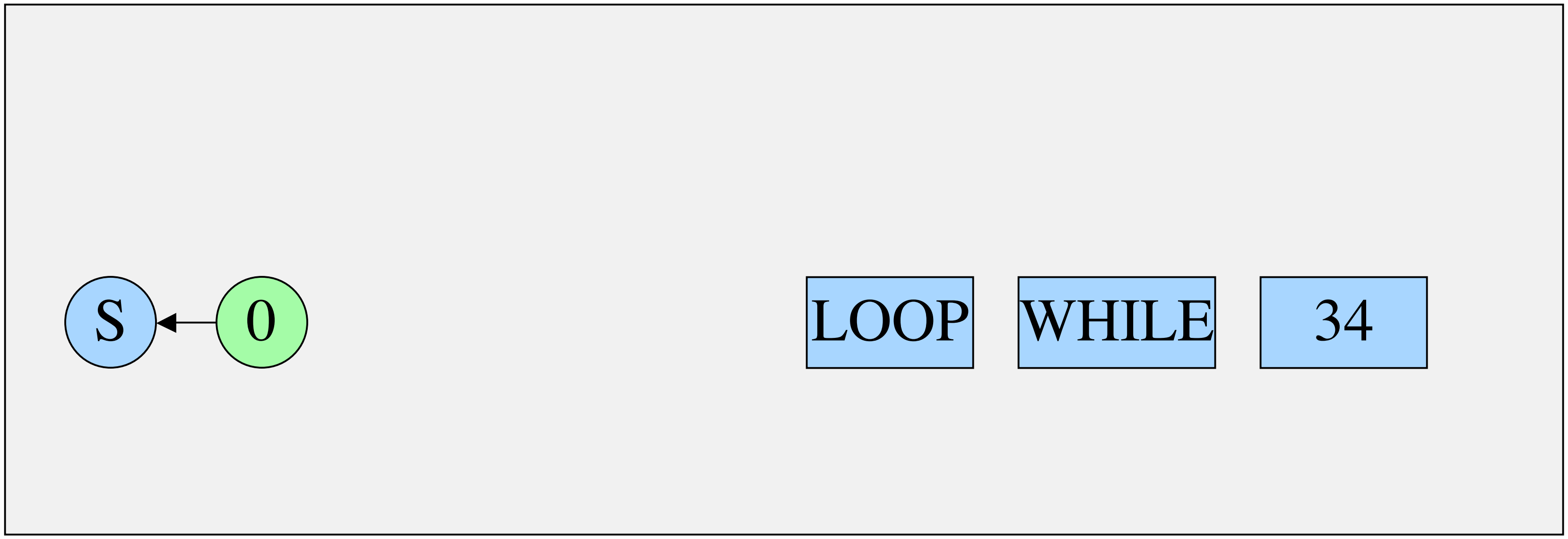

State 0
$S' \rightarrow \bullet\, b_L, \$$
$b_L \rightarrow \bullet\, LOOP_L\ d_w\ END_L, \$$
$loop_L \rightarrow \bullet\, , WHILE_w$
$loop_L \rightarrow \bullet\, LOOP_L, WHILE_w$
b_L
State 9
$S' \rightarrow b_L \bullet\, , \$$
$loop_L$
State 4
$loop_L \rightarrow LOOP_L \bullet\, , WHILE_w$
$LOOP_L$
State 1
$b_L \rightarrow LOOP_L \bullet\, d_w\ END_L, \$$
$d_w \rightarrow \bullet\, WHILE_w\ NUM_w\ do_w, END_L$
d_w
State 5
$b_L \rightarrow LOOP_L\ d_w \bullet\, END_L, \$$
END_L
State 6
$b_L \rightarrow LOOP_L\ d_w\ END_L \bullet\, , \$$
$WHILE_w$
State 2
$d_w \rightarrow WHILE_w \bullet\, NUM_w\ do_w, END_L$
State 7
$d_w \rightarrow WHILE_w\ NUM_w\ DO_w \bullet\, , END_L$
NUM_w
DO_w
State 3
$d_w \rightarrow WHILE_w\ NUM_w \bullet\, DO_w, END_L$
$do_w \rightarrow \bullet\, , END_L$
$do_w \rightarrow \bullet\, DO_w, END_L$
do_w
State 8
$do_w \rightarrow DO_w \bullet\, , END_L$

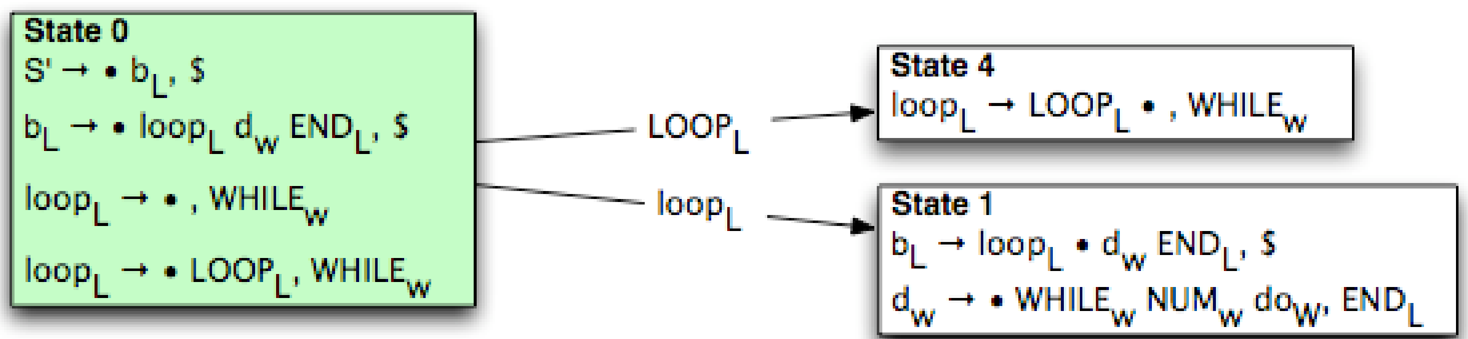

State 0
S' → • b_L, $
b_L → • LOOP_L d_W END_L, $
loop_L → • , WHILE_W
loop_L → • LOOP_L, WHILE_W
b_L
State 9
S' → b_L • , $
loop_L
State 4
loop_L → LOOP_L • , WHILE_W
LOOP_L
State 1
b_L → LOOP_L • d_W END_L, $
d_W → • WHILE_W NUM_W do_W, END_L
d_W
State 5
b_L → LOOP_L d_W • END_L, $
END_L
State 6
b_L → LOOP_L d_W END_L • , $
WHILE_W
State 2
d_W → WHILE_W • NUM_W do_W, END_L
NUM_W
State 7
d_W → WHILE_W NUM_W DO_W • , END_L
DO_W
State 3
d_W → WHILE_W NUM_W • DO_W, END_L
do_W → • , END_L
do_W → • DO_W, END_L
do_W
State 8
do_W → DO_W • , END_L

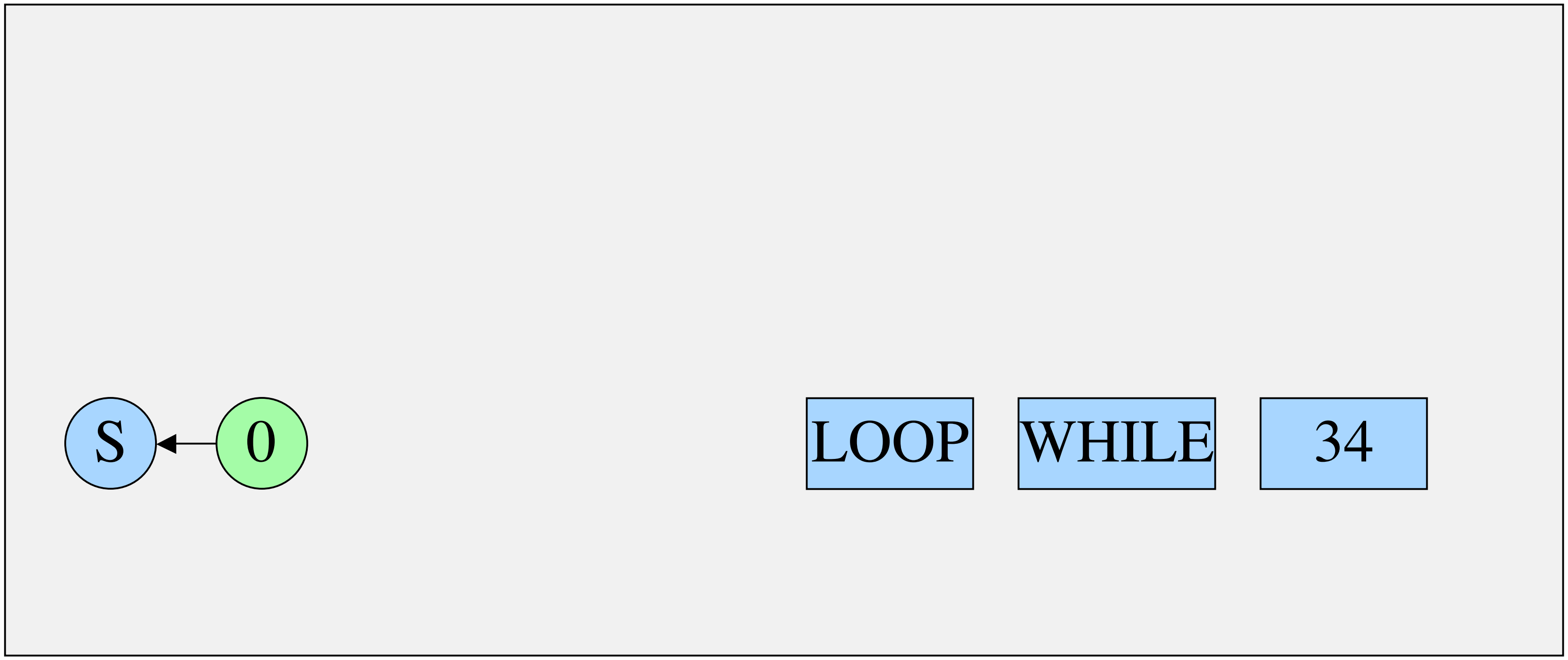

State 0
S' → • b_L, $
b_L → • LOOP_L d_W END_L, $
loop_L → • , WHILE_W
loop_L → • LOOP_L, WHILE_W
b_L
State 9
S' → b_L • , $
loop_L
State 4
loop_L → LOOP_L • , WHILE_W
LOOP_L
State 1
b_L → LOOP_L • d_W END_L, $
d_W → • WHILE_W NUM_W do_W, END_L
d_W
State 5
b_L → LOOP_L d_W • END_L, $
END_L
State 6
b_L → LOOP_L d_W END_L • , $
WHILE_W
State 2
d_W → WHILE_W • NUM_W do_W, END_L
NUM_W
State 3
d_W → WHILE_W NUM_W • DO_W, END_L
do_W → • , END_L
do_W → • DO_W, END_L
DO_W
State 7
d_W → WHILE_W NUM_W DO_W • , END_L
do_W
State 8
do_W → DO_W • , END_L


State 0
$S' \rightarrow \cdot\, b_L, \$$
$b_L \rightarrow \cdot\, LOOP_L\ d_W\ END_L, \$$
$loop_L \rightarrow \cdot\, , WHILE_W$
$loop_L \rightarrow \cdot\, LOOP_L, WHILE_W$
b_L
State 9
$S' \rightarrow b_L \bullet , \$$
$loop_L$
State 4
$loop_L \rightarrow LOOP_L \bullet , WHILE_W$
$LOOP_L$
State 1
$b_L \rightarrow LOOP_L \bullet d_W\ END_L, \$$
$d_W \rightarrow \bullet\, WHILE_W\ NUM_W\ do_W, END_L$
d_W
State 5
$b_L \rightarrow LOOP_L\ d_W \bullet END_L, \$$
END_L
State 6
$b_L \rightarrow LOOP_L\ d_W\ END_L \bullet , \$$
$WHILE_W$
State 2
$d_W \rightarrow WHILE_W \bullet NUM_W\ do_W, END_L$
State 7
$d_W \rightarrow WHILE_W\ NUM_W\ DO_W \bullet , END_L$
NUM_W
DO_W
State 3
$d_W \rightarrow WHILE_W\ NUM_W \cdot DO_W, END_L$
$do_W \rightarrow \cdot\, , END_L$
$do_W \rightarrow \cdot\, DO_W, END_L$
do_W
State 8
$do_W \rightarrow DO_W \bullet , END_L$

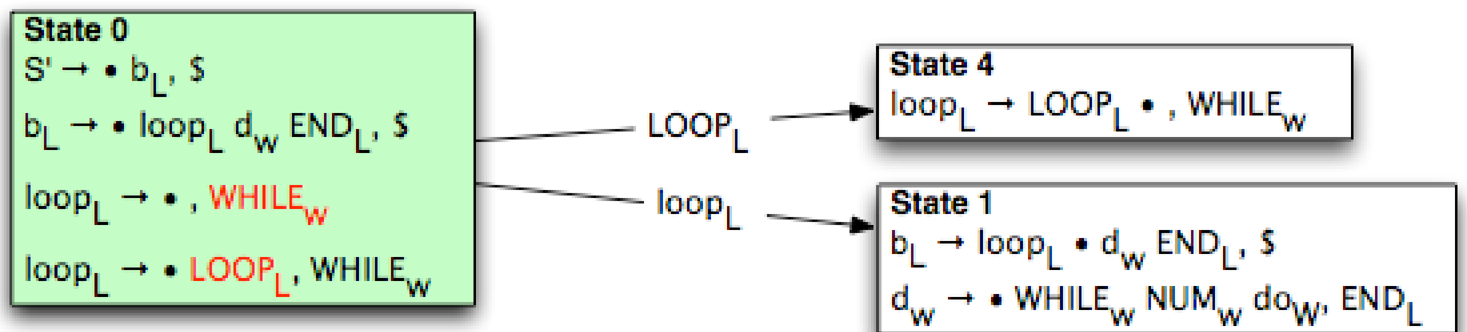
Parsing Embedded Languages

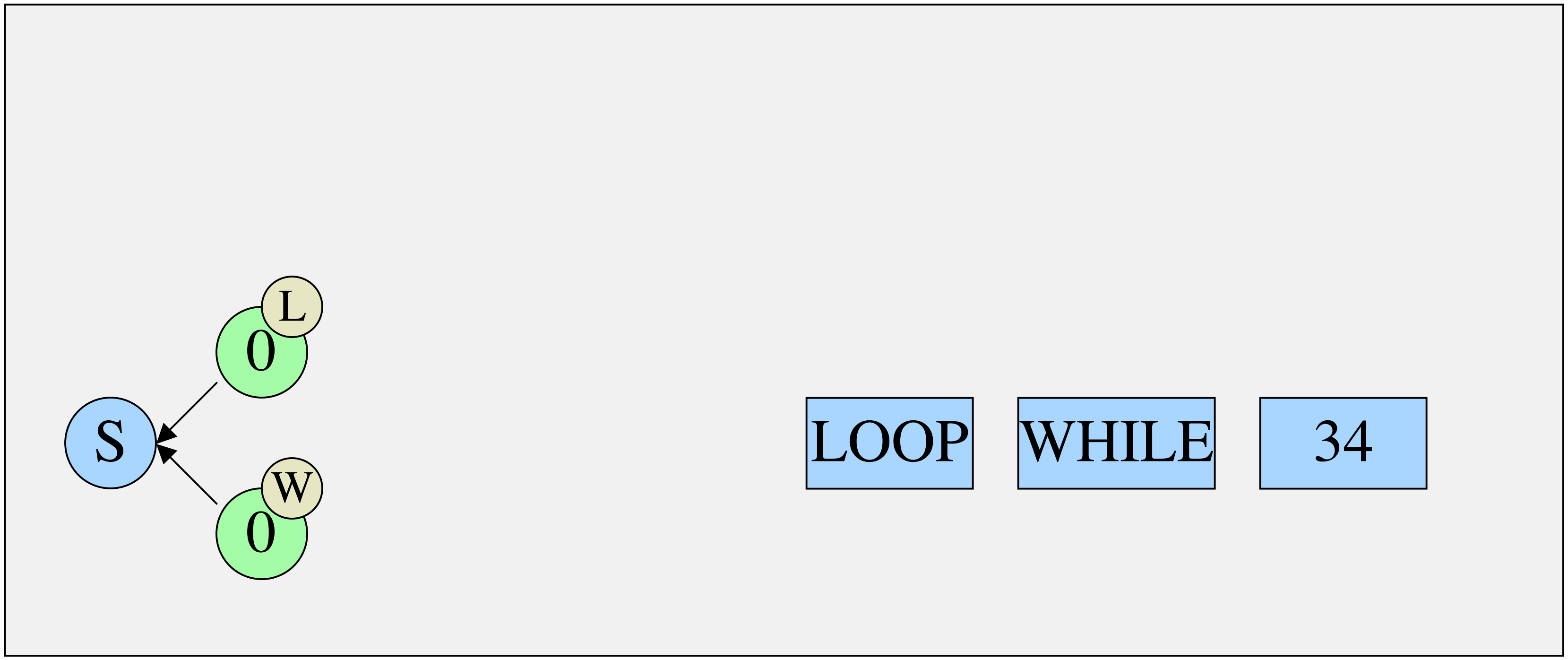




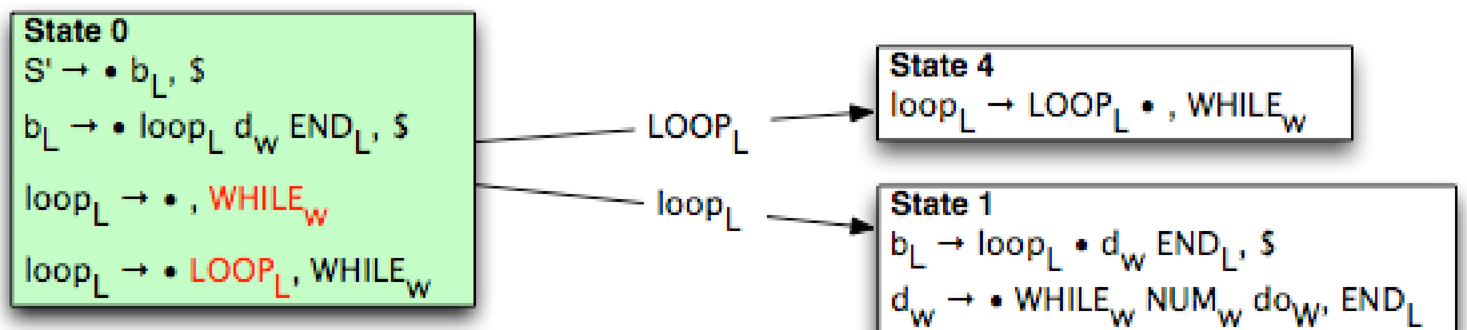


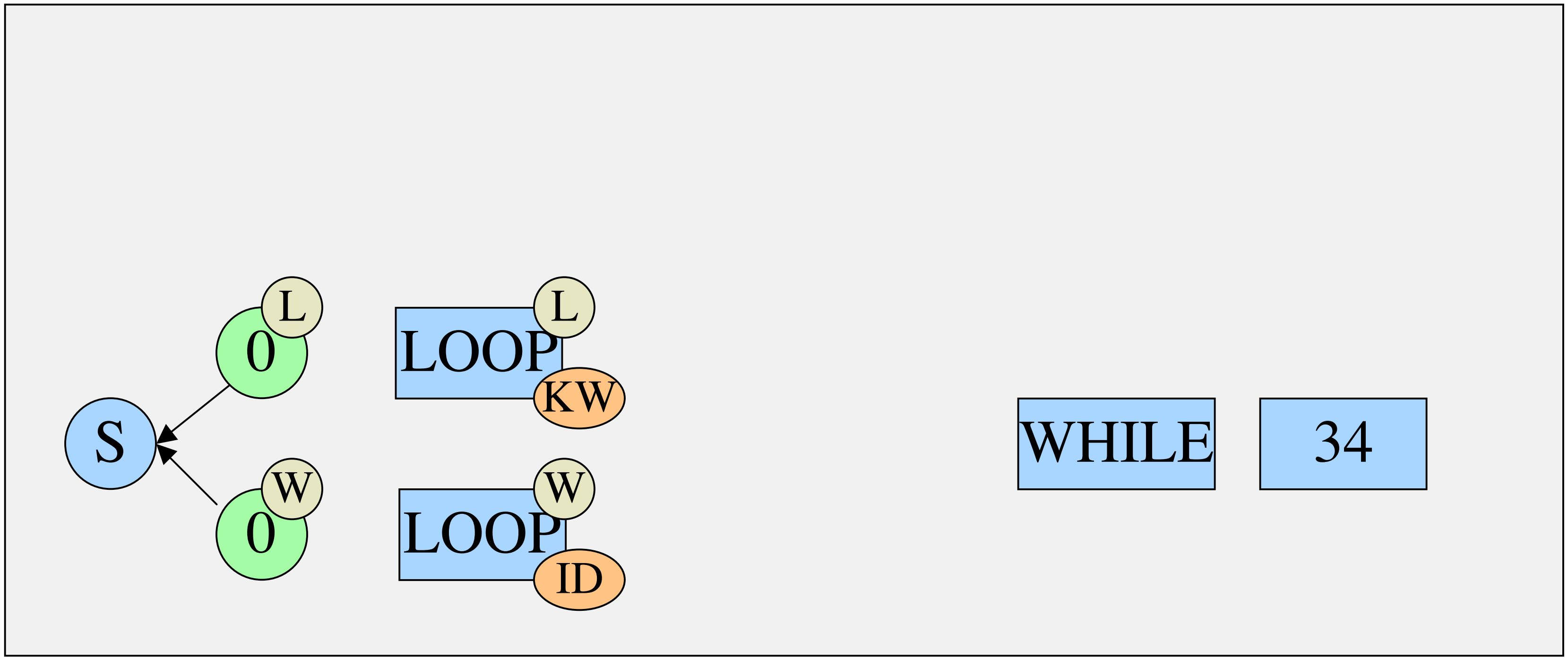
Current parse state has ambiguous lexical language





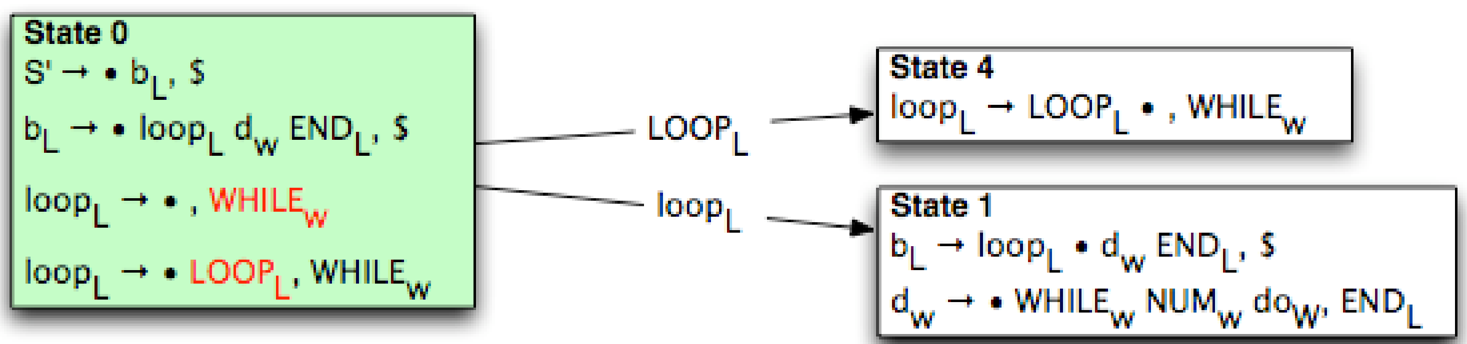
XGLR Extension: Fork parsers, assign one to each lexical language

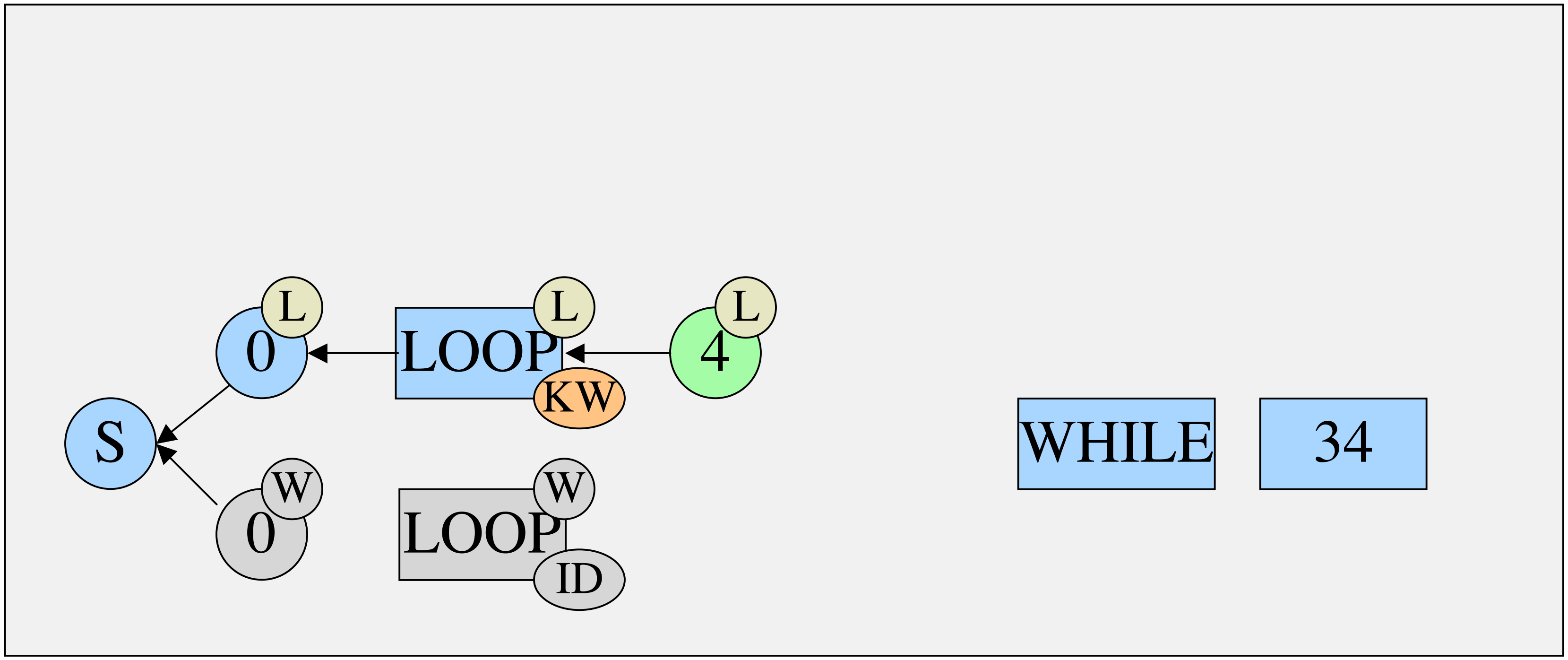




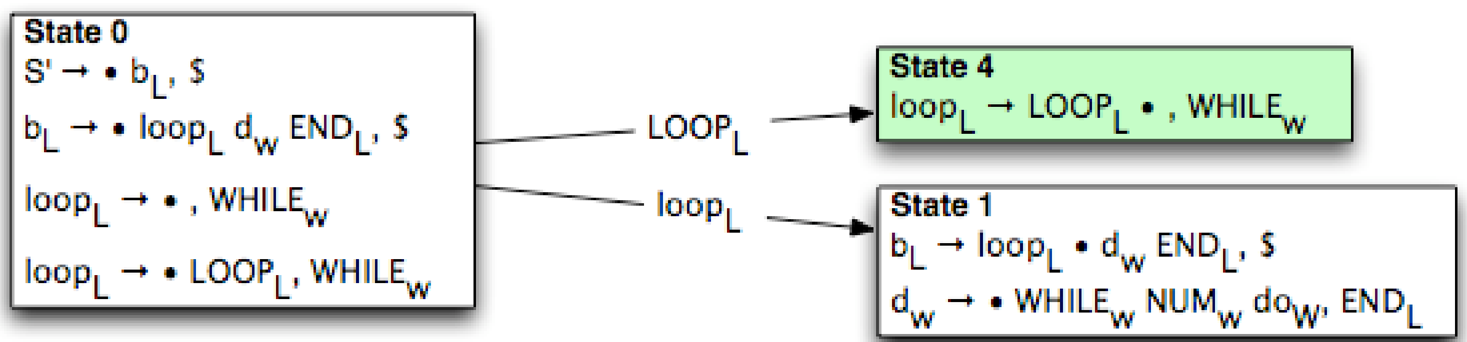
XGLR Extension: Single spelling, Multiple lexical categories

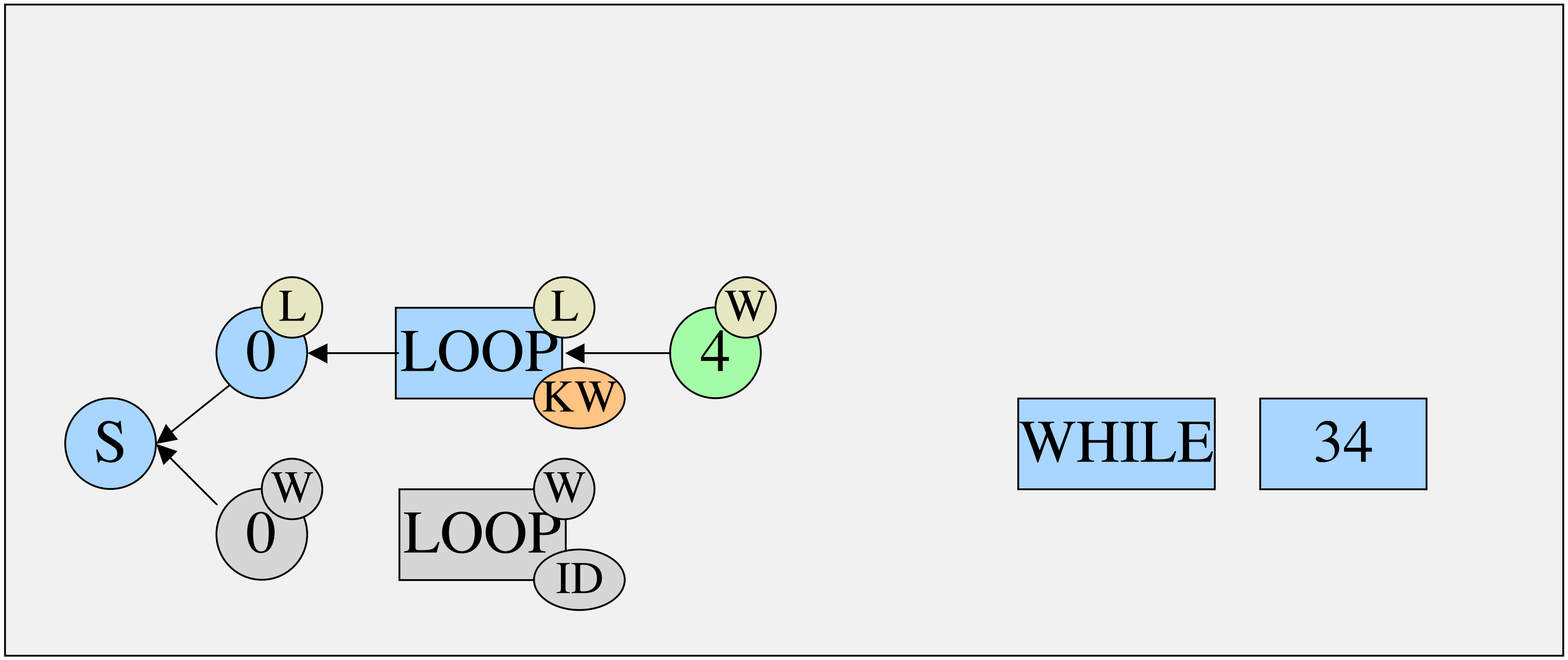
Lex lookahead both in language L and W



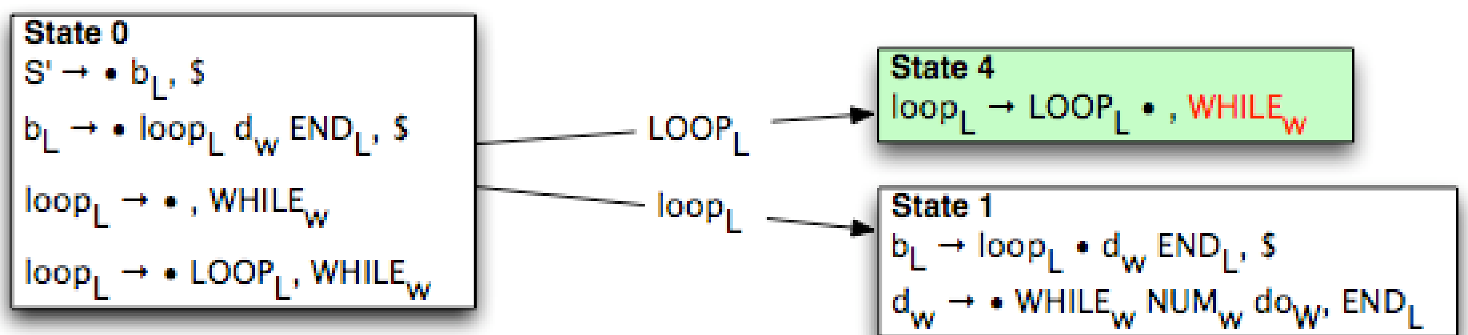


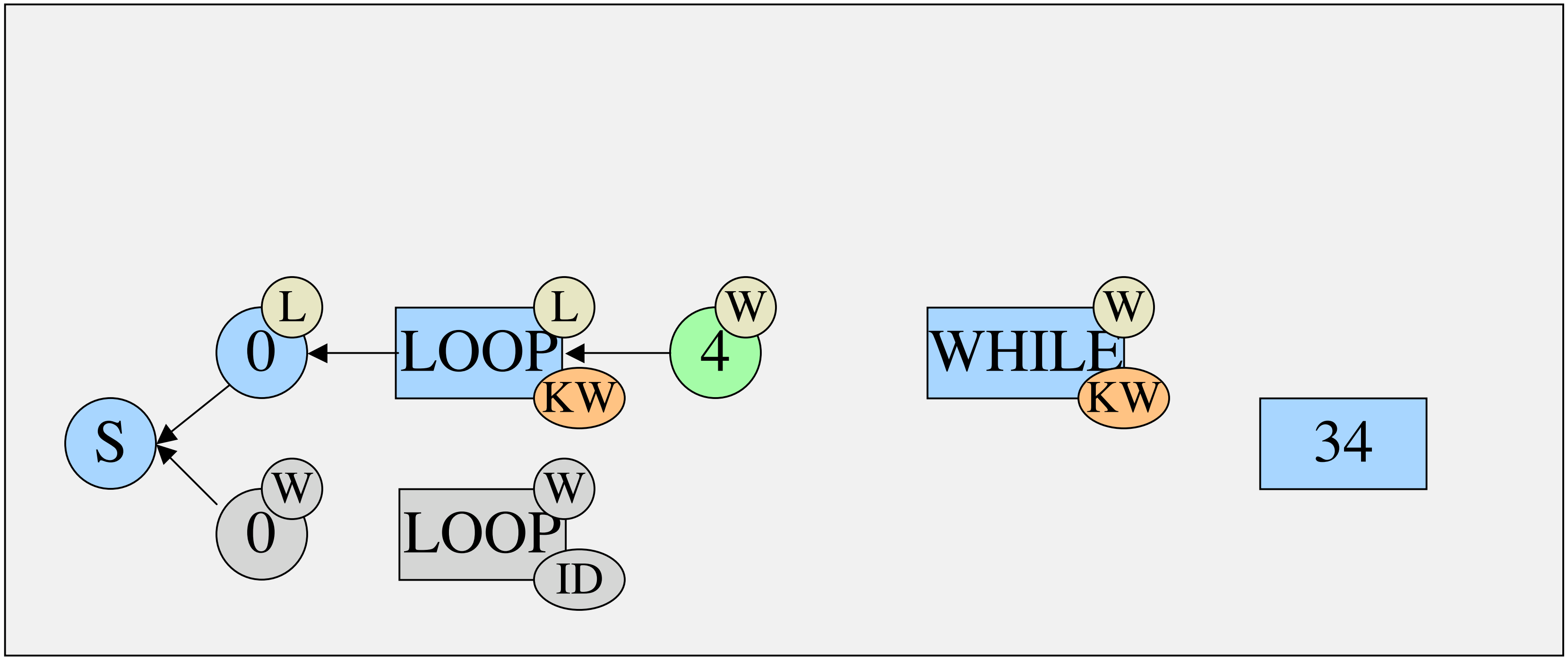
Only $LOOP_L$ is valid lookahead, and is shifted



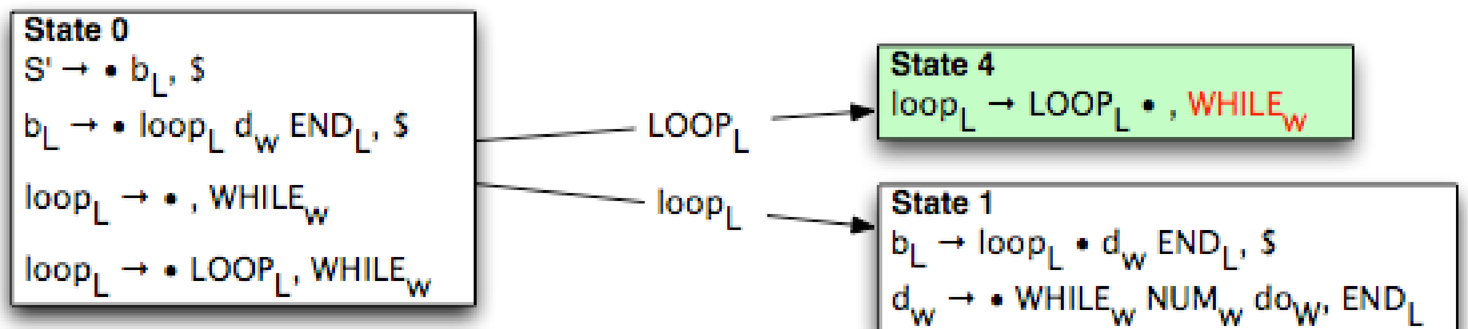


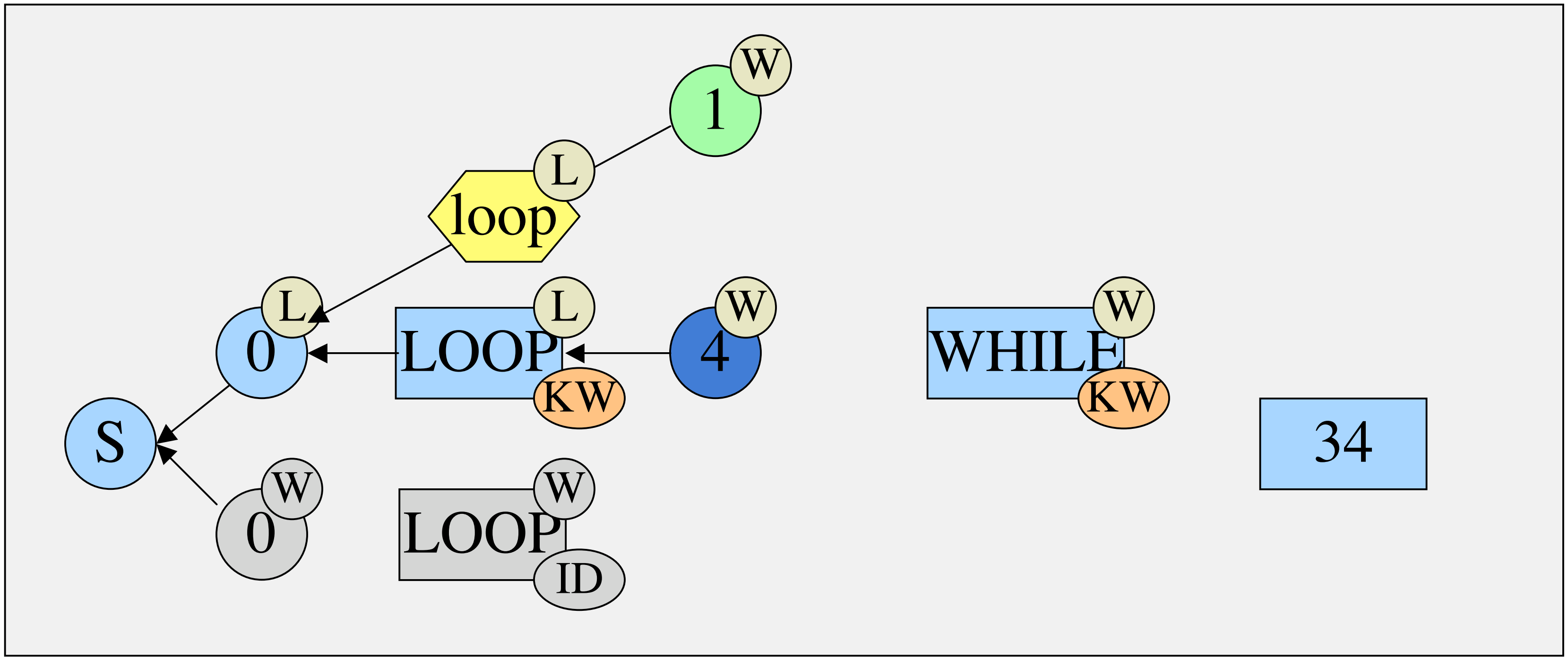
XGLR Extension: State 4 has lexer lookaheads only in language W



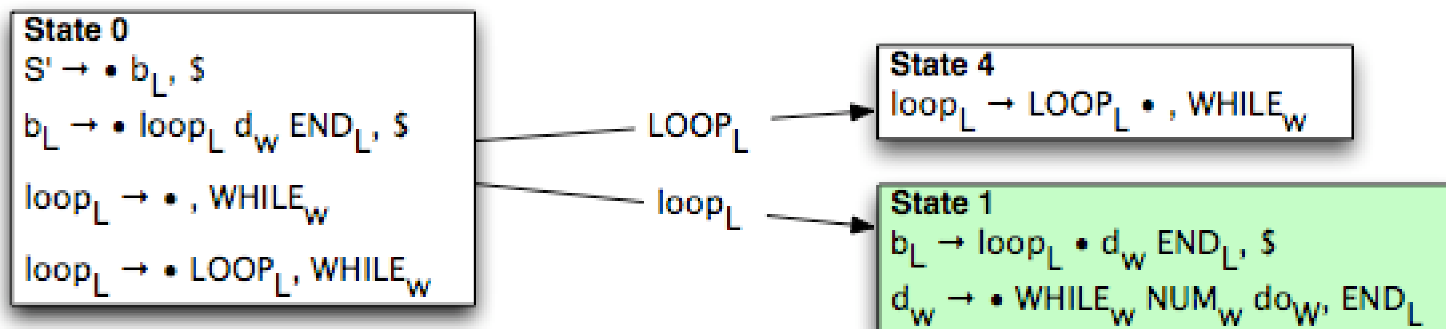


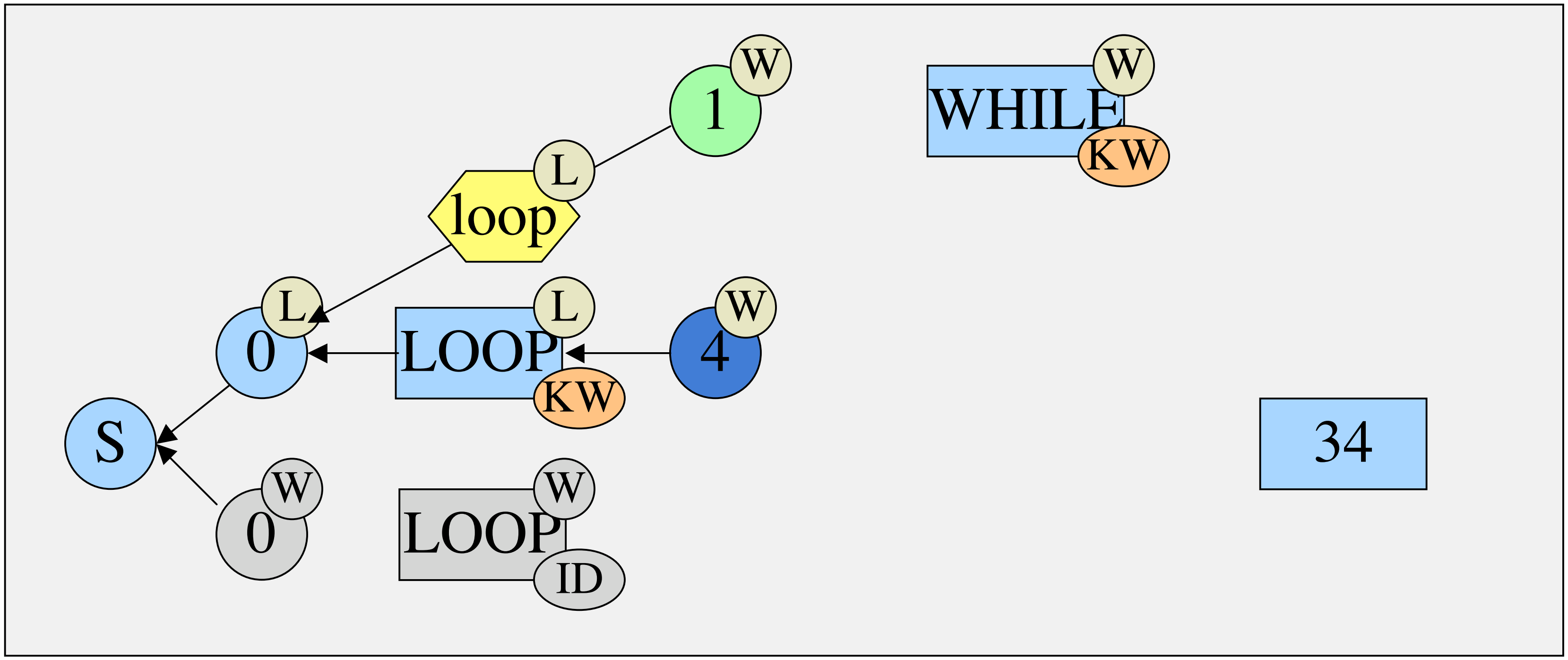
Lex lookahead in language W

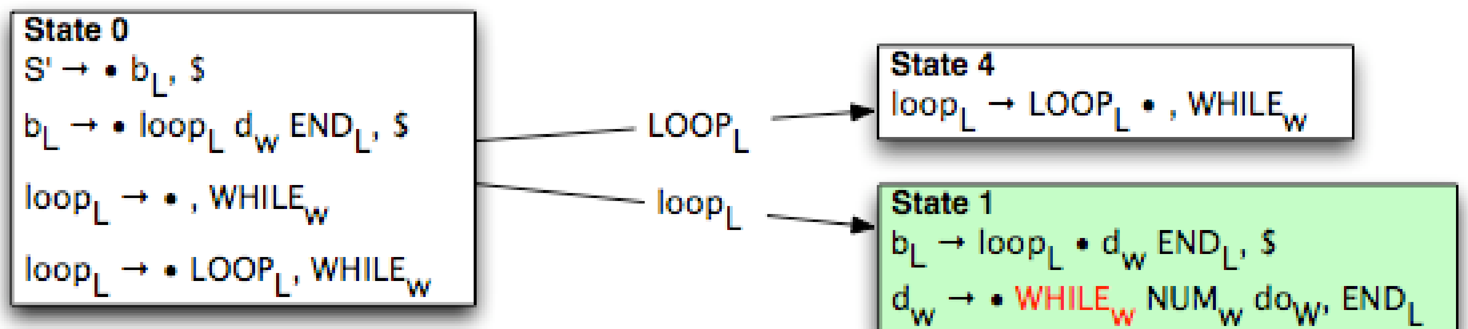


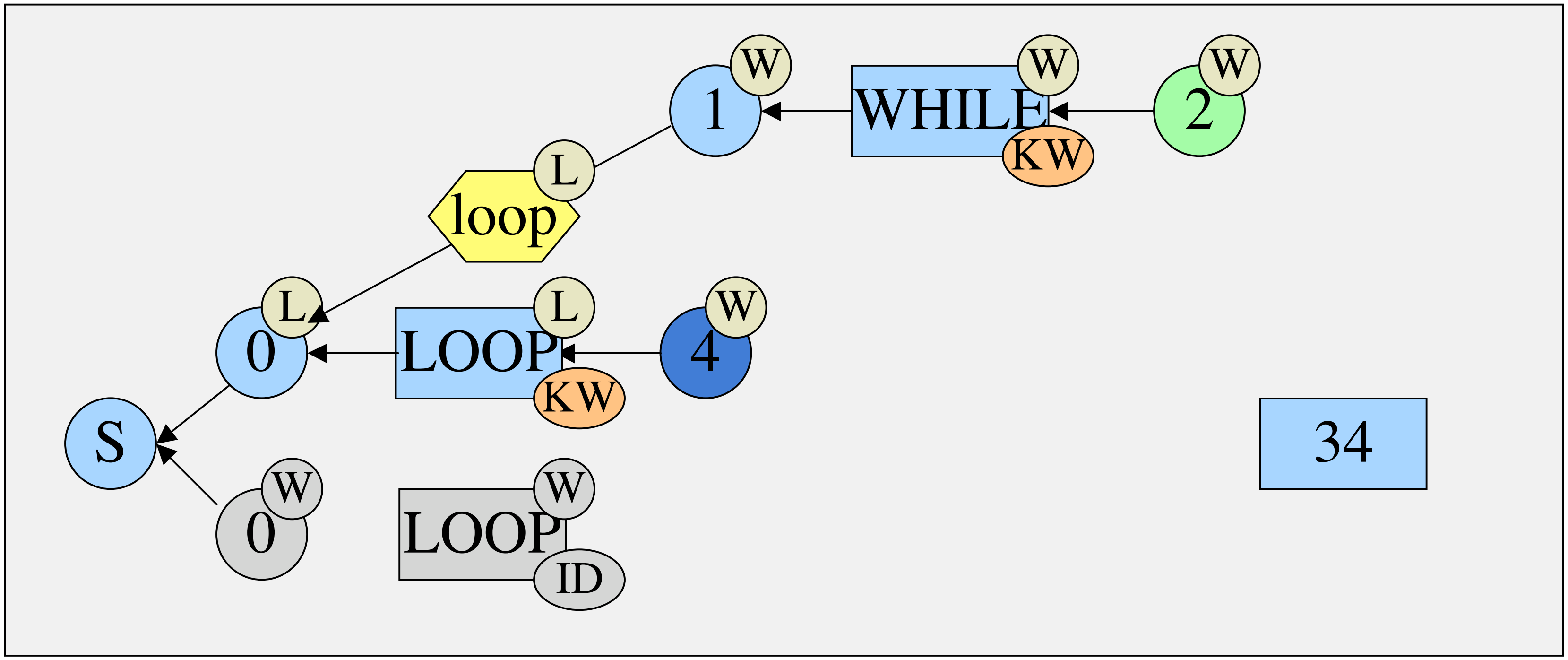


REDUCE by rule 2 and GOTO state 1

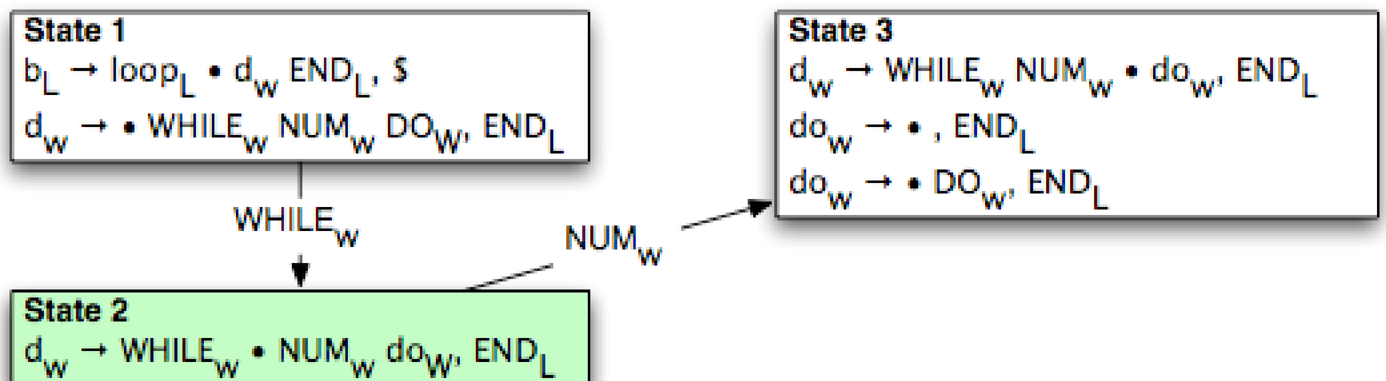


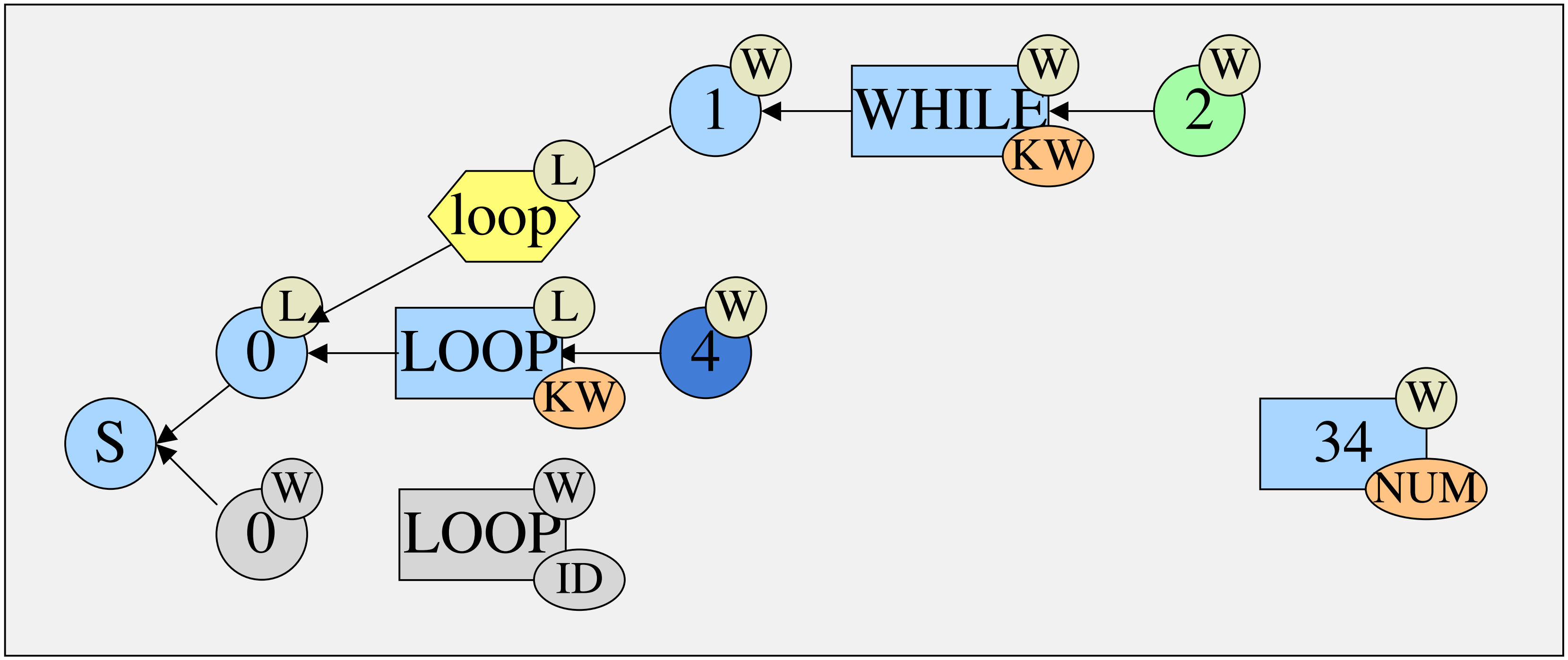

S
0
L
0
W
loop
L
1
W
LOOP
L
KW
4
W
LOOP
W
ID
WHILE
W
KW
34


State 0
$S' \to \bullet\ b_L,\ \$$
$b_L \to \bullet\ loop_L\ d_W\ END_L,\ \$$
$loop_L \to \bullet\ ,\ WHILE_W$
$loop_L \to \bullet\ LOOP_L,\ WHILE_W$
$LOOP_L$
$loop_L$
State 4
$loop_L \to LOOP_L\ \bullet\ ,\ WHILE_W$
State 1
$b_L \to loop_L\ \bullet\ d_W\ END_L,\ \$$
$d_W \to \bullet\ WHILE_W\ NUM_W\ do_W,\ END_L$

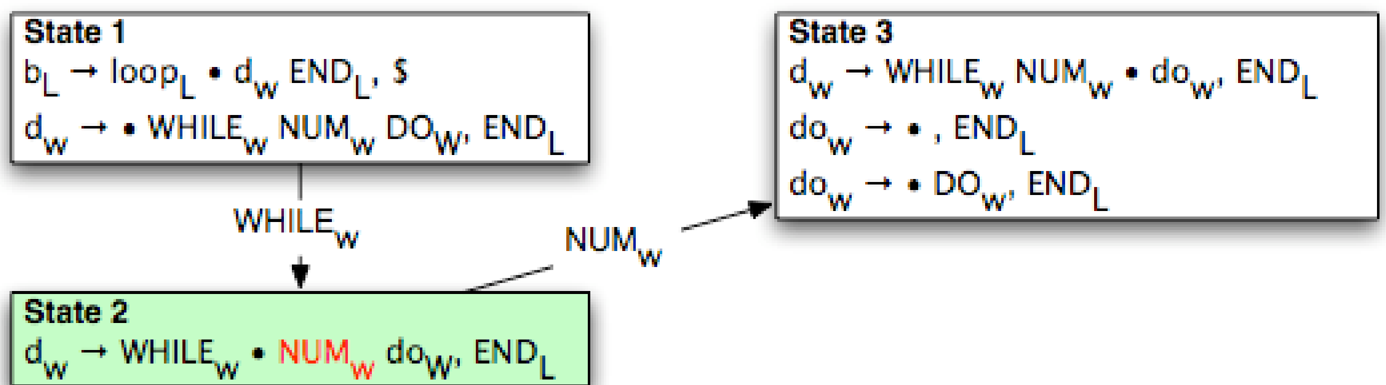


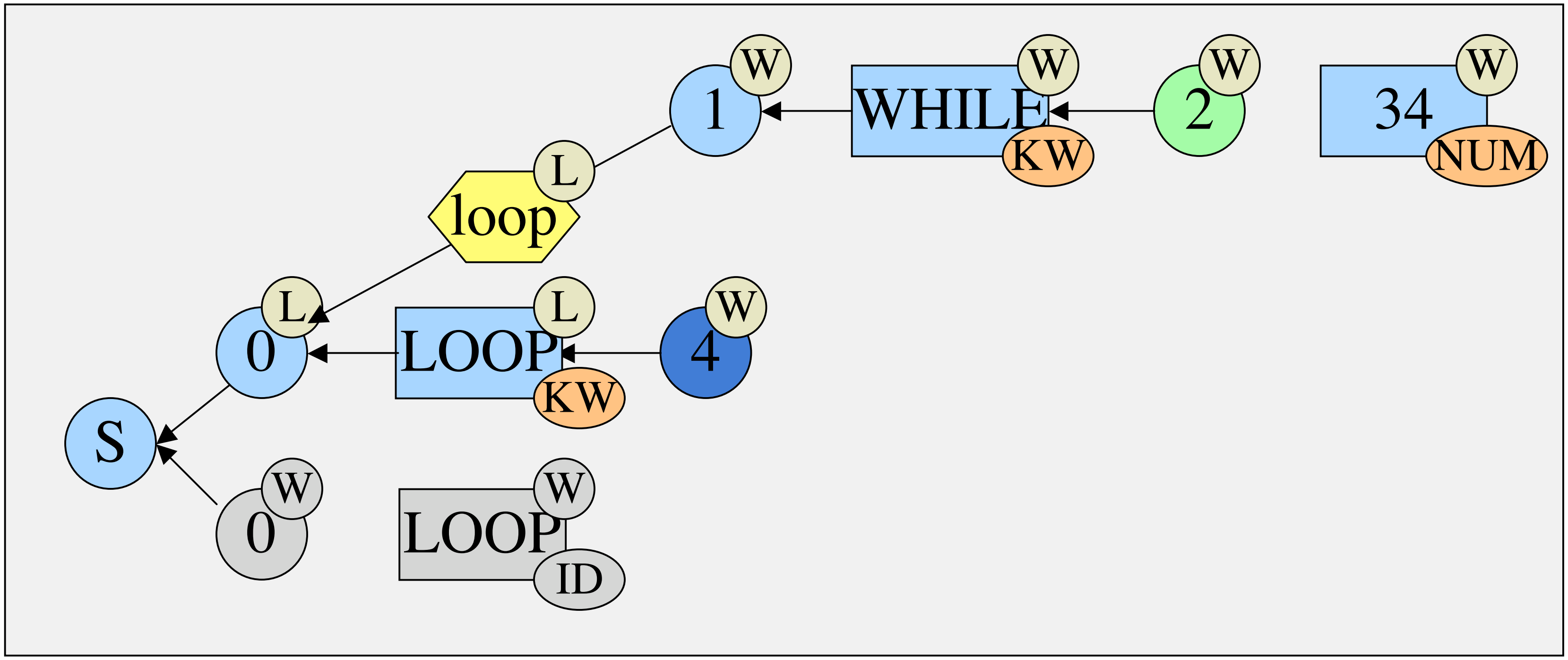
Shift into state 2

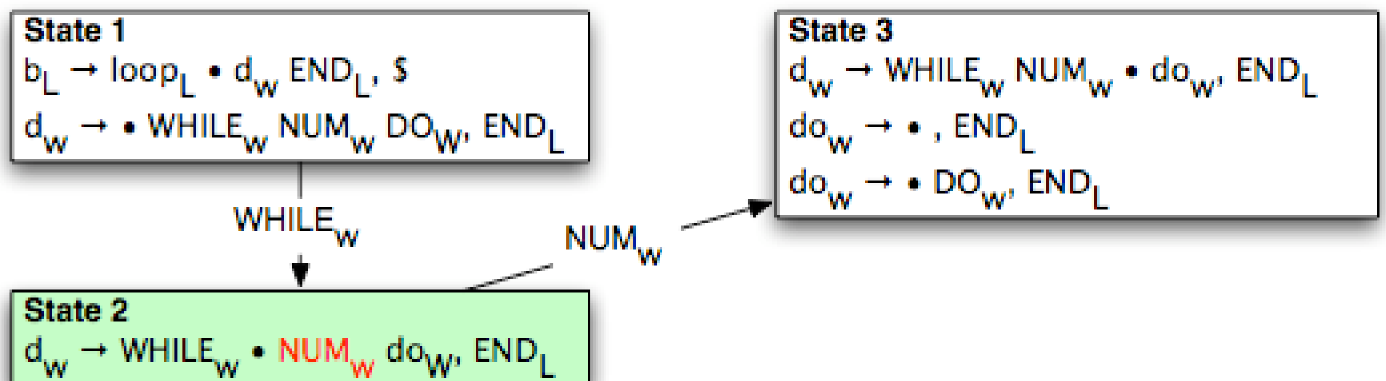


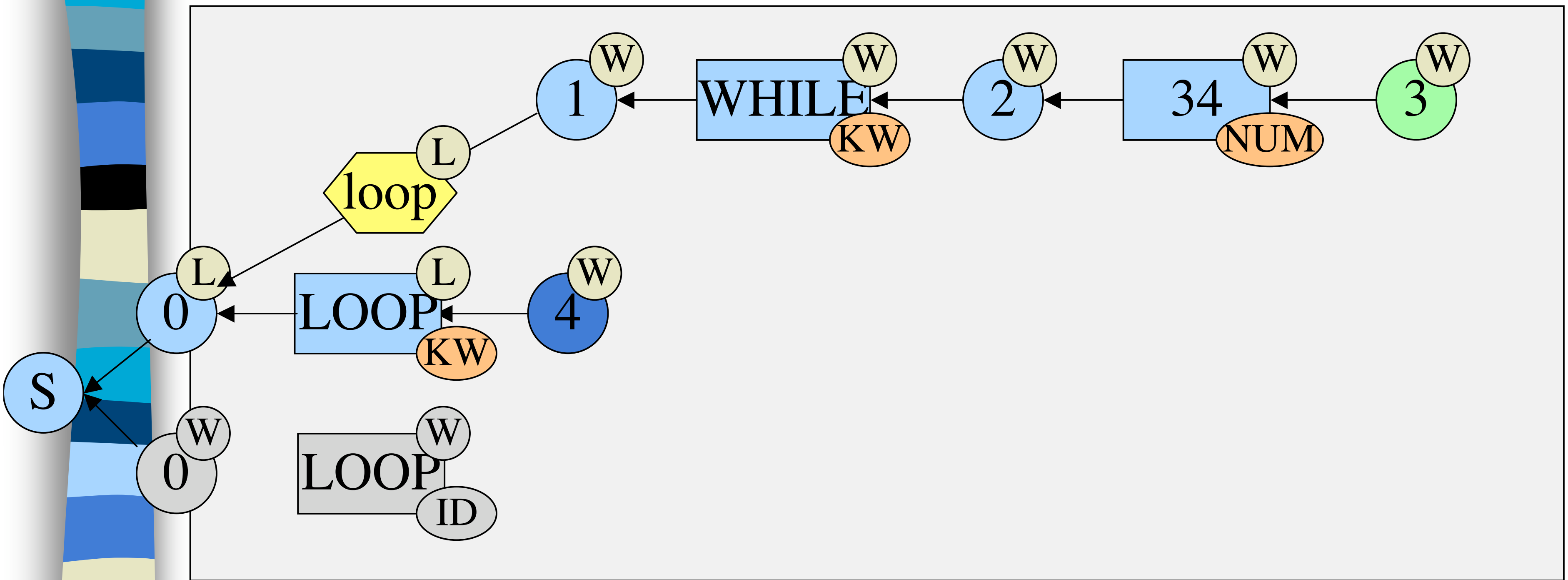


XGLR Extension: Lex lookahead in language W

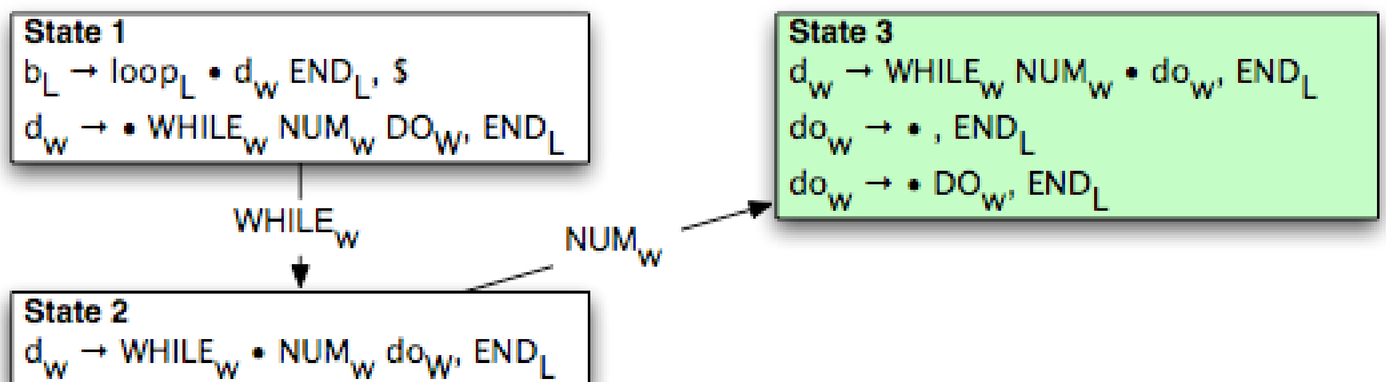


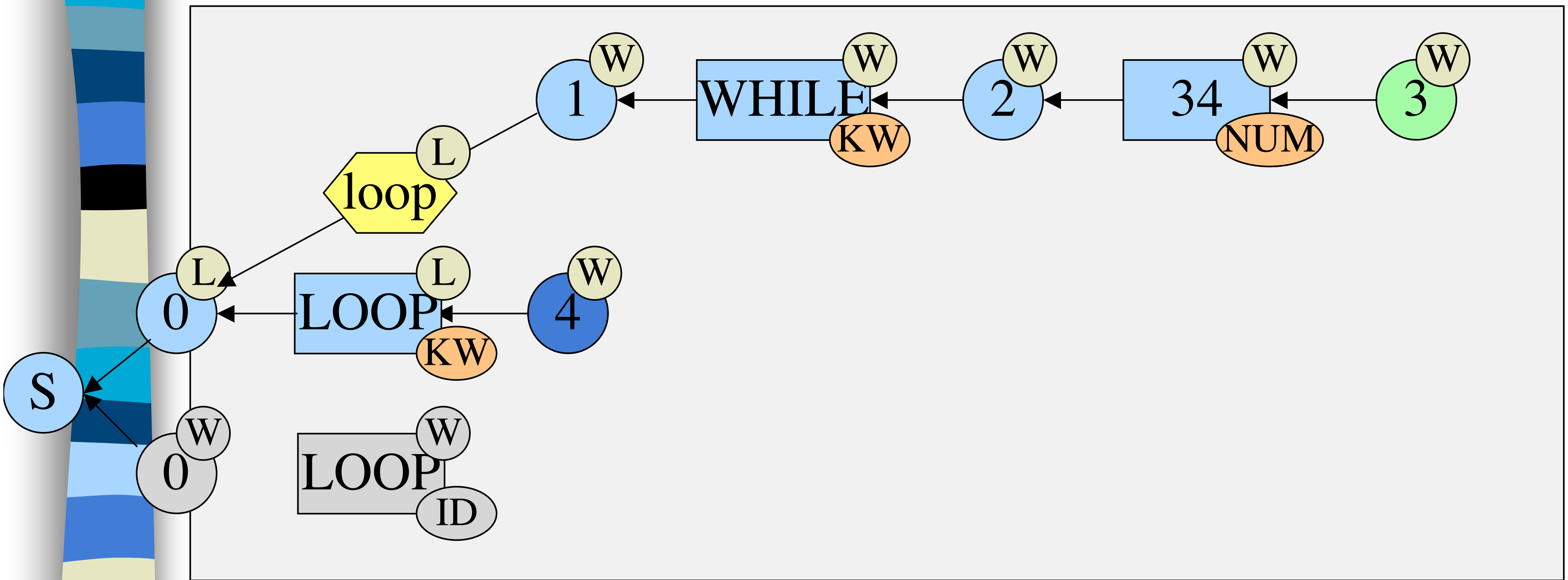

W
WHILE
W
KW
W
2
W
34
NUM
1
L
loop
L
0
L
LOOP
KW
W
4
S
W
0
W
LOOP
ID


State 1
$b_L \rightarrow loop_L \bullet d_w\ END_L, \$$
$d_w \rightarrow \bullet\ WHILE_w\ NUM_w\ DO_W, END_L$
WHILE$_w$
State 2
$d_w \rightarrow WHILE_w \bullet NUM_w\ do_W, END_L$
NUM$_w$
State 3
$d_w \rightarrow WHILE_w\ NUM_w \bullet do_W, END_L$
$do_w \rightarrow \bullet\ , END_L$
$do_w \rightarrow \bullet\ DO_W, END_L$

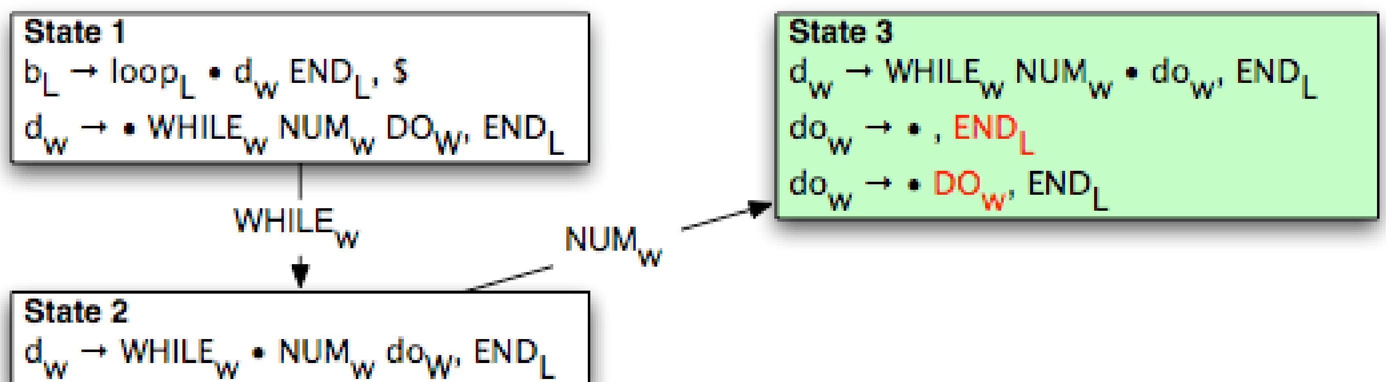


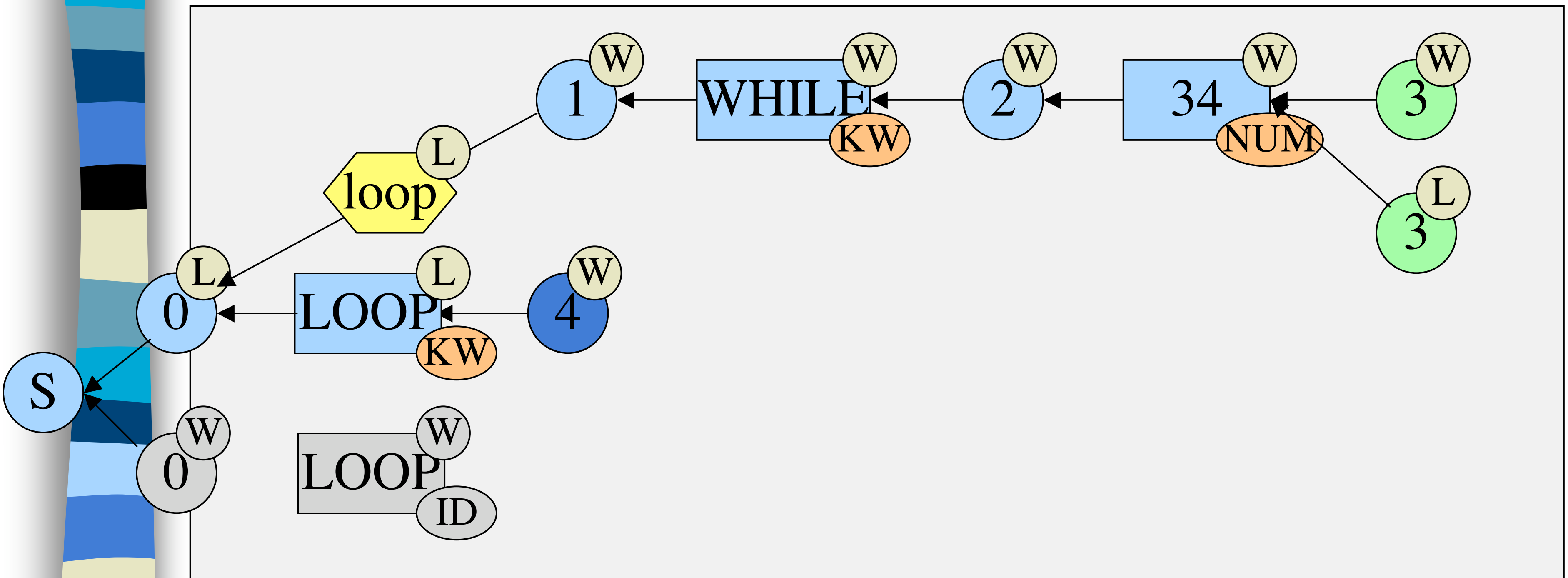
Shift into state 3





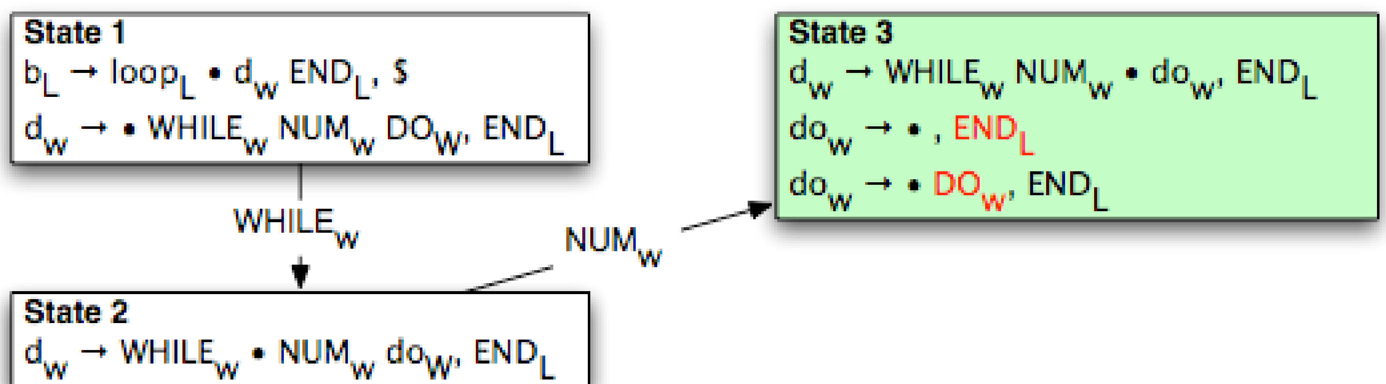
Shift into state 3, which has ambiguous lexical language





XGLR Extension: Single spelling, Multiple lexical categories

Fork parsers, assign one to each lexical language



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

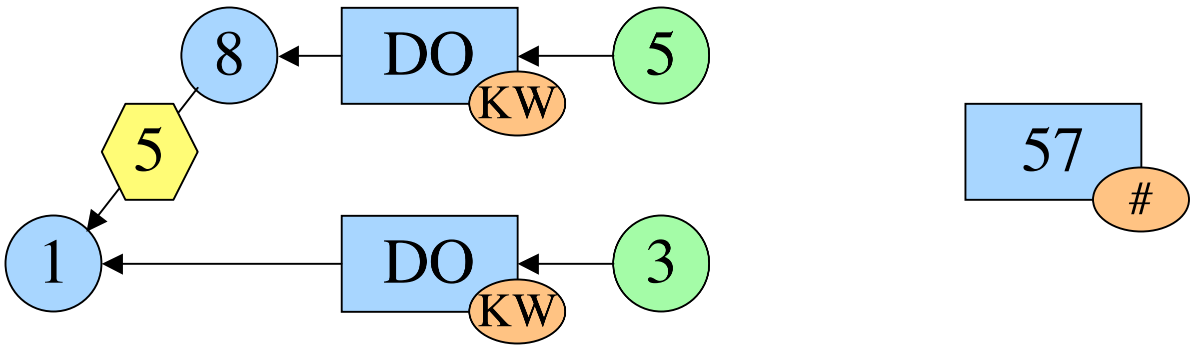
- Many GLR programming language specs have finite, few ambiguities
- XGLR language specs *also* have finite, but slightly more, ambiguities
  - Lexical ambiguity due to ambiguous input does result in more ambiguous parse forests

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

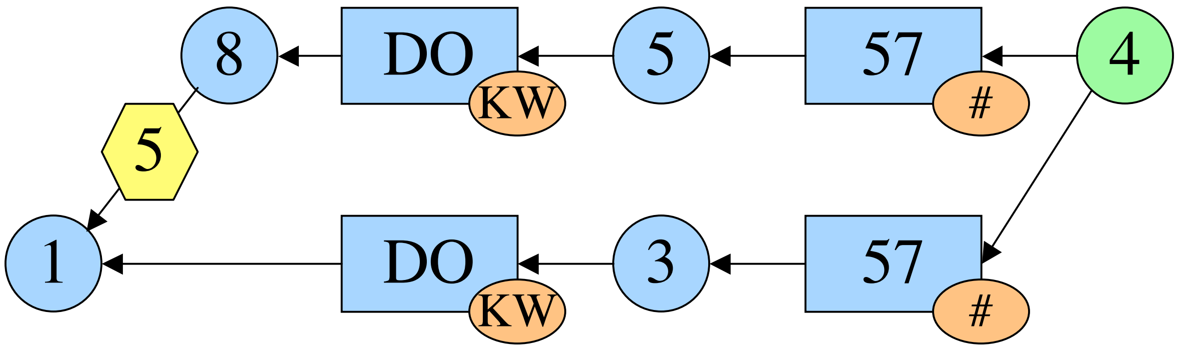
Parser Merging

- GLR: Parsers merge when in same parse state



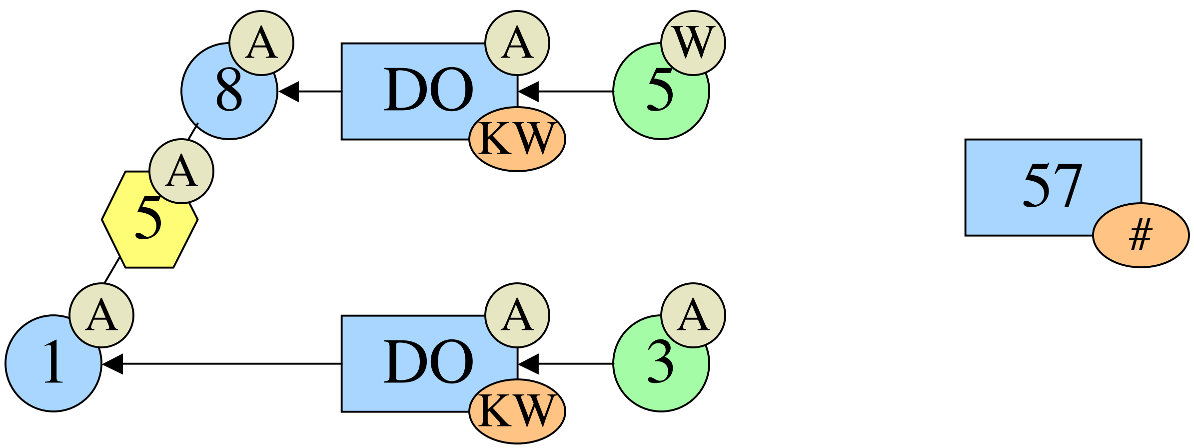
Parser Merging

- GLR: Parsers merge when in same parse state



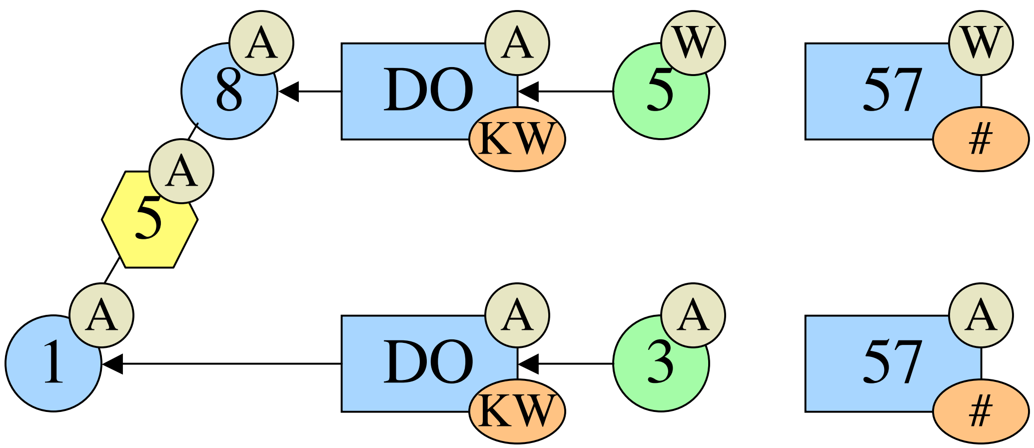
Parser Merging

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



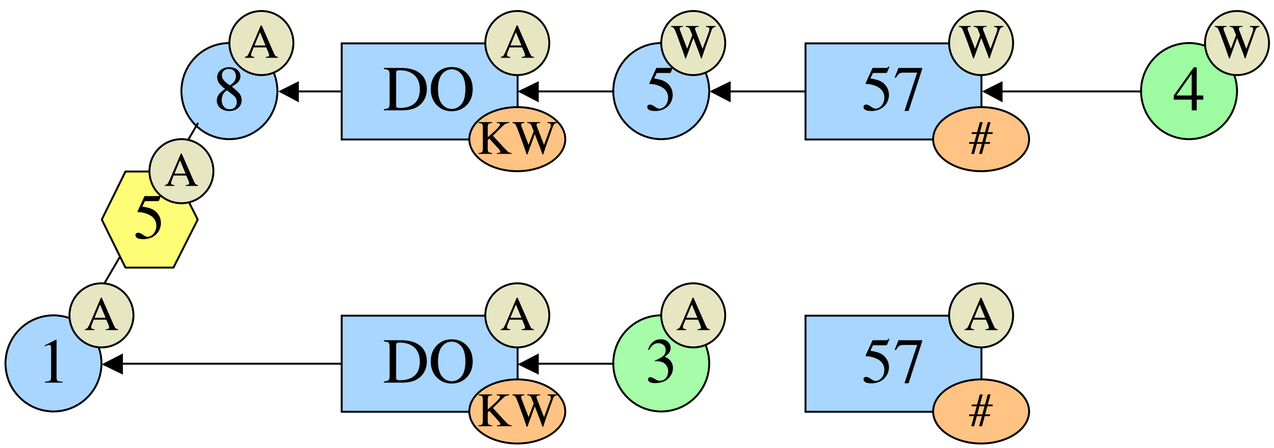
Parser Merging

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



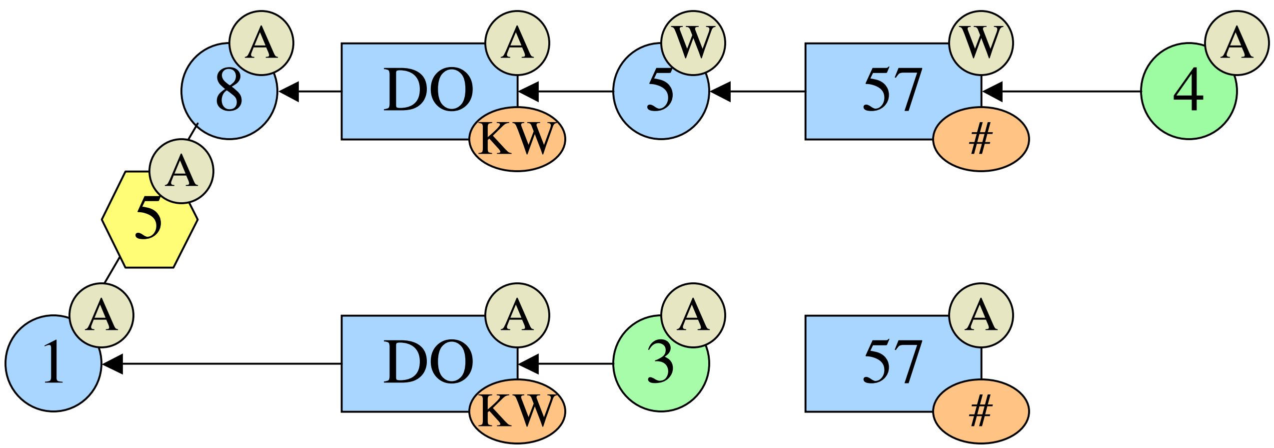
Parser Merging

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



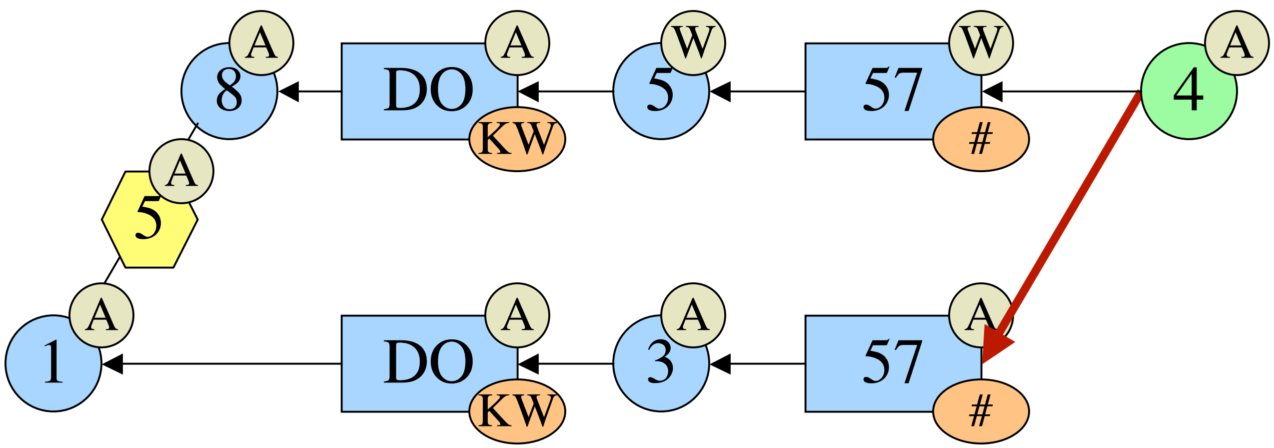
Parser Merging

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



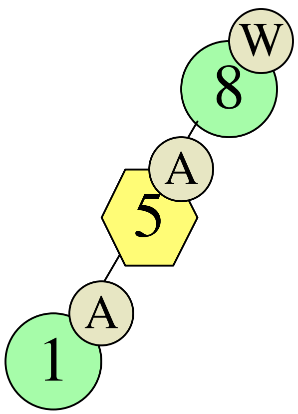
Parser Merging

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



Out of Sync Parsers

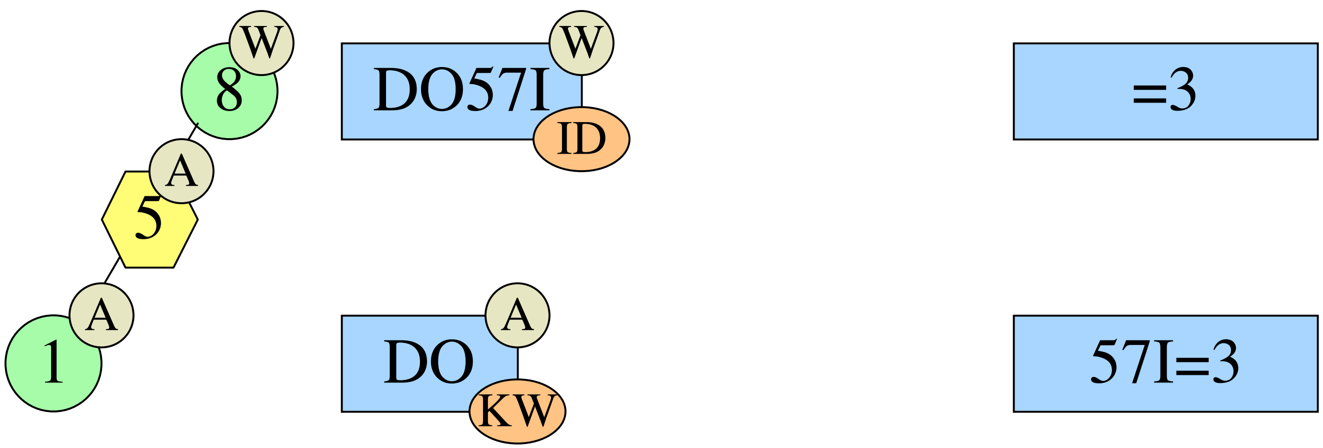
- XGLR: Parsers merge when in same parse state and same lexical state *and same input position*



DO57I=3

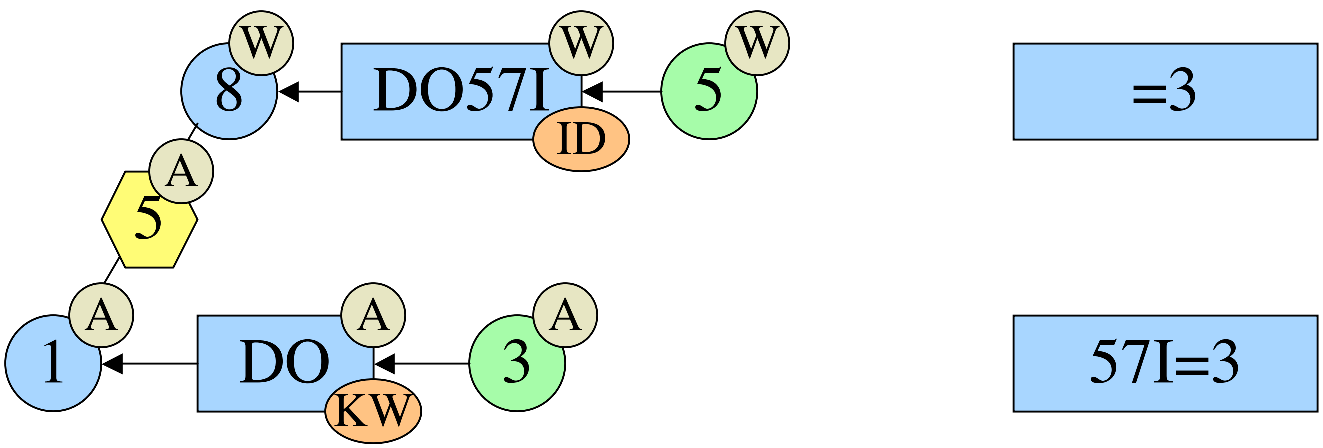
Out of Sync Parsers

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



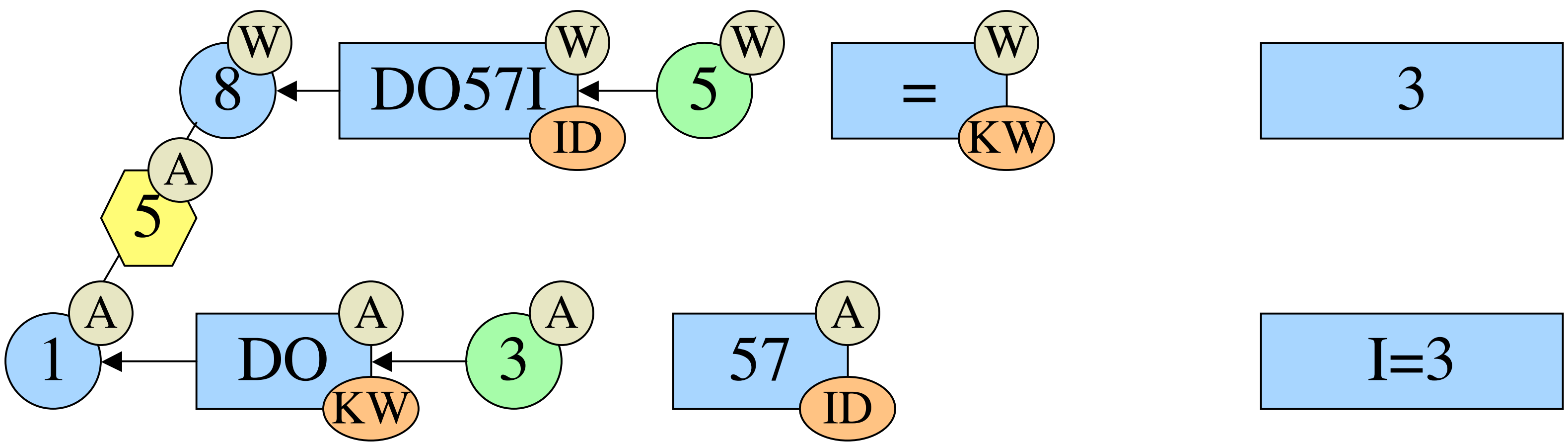
Out of Sync Parsers

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



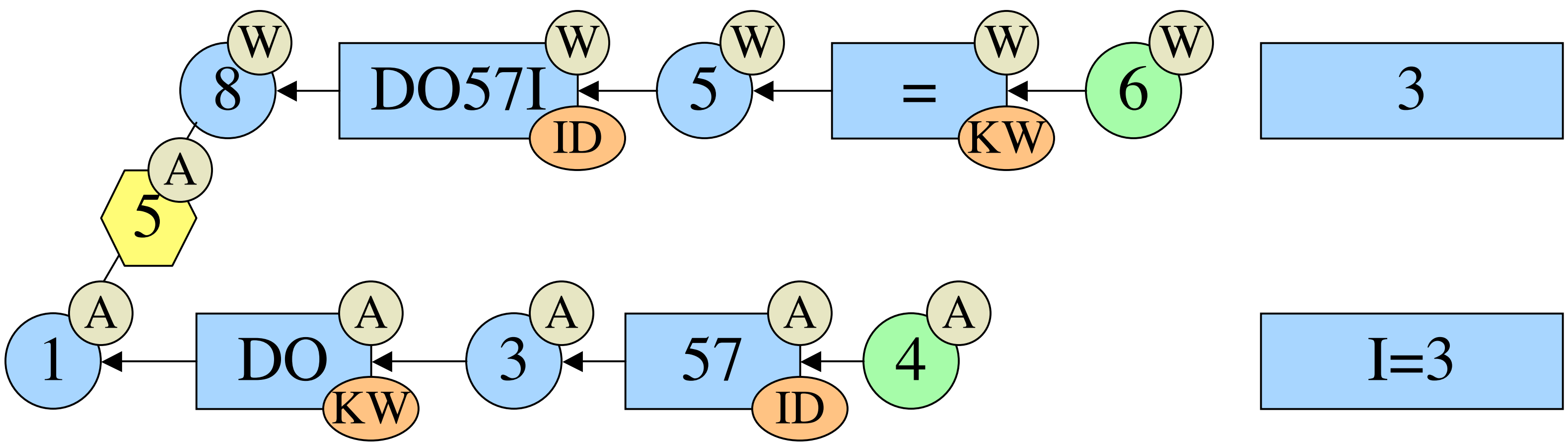
Out of Sync Parsers

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



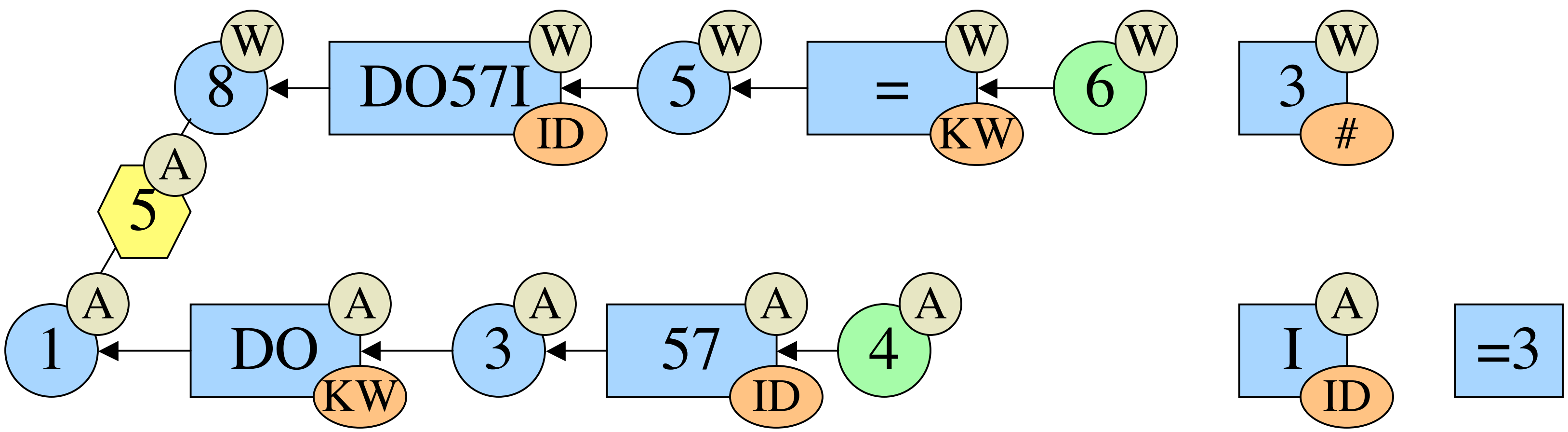
Out of Sync Parsers

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



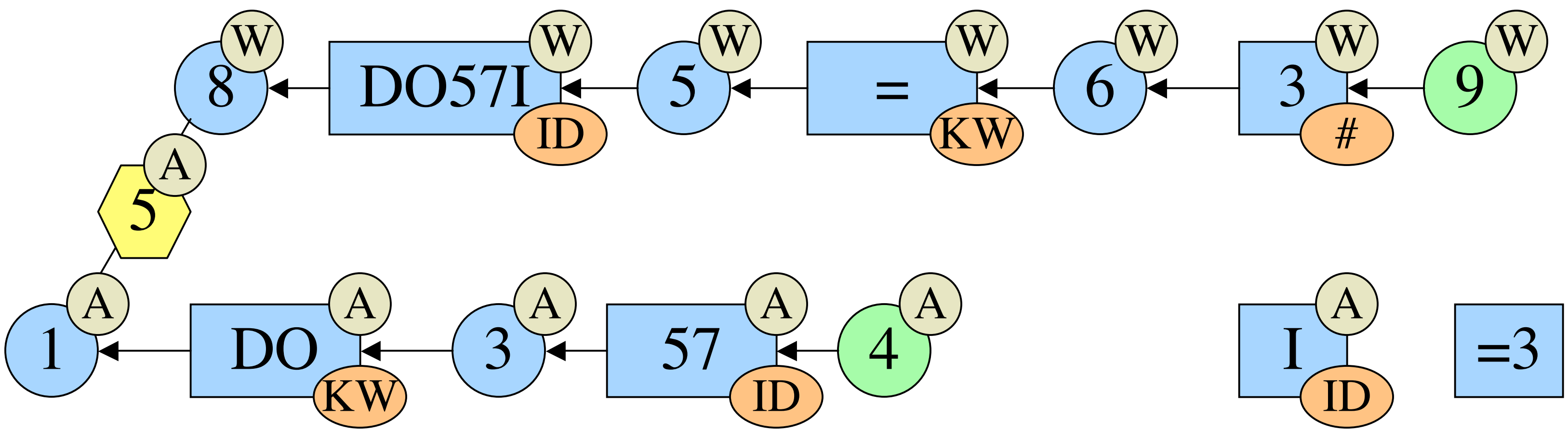
Out of Sync Parsers

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



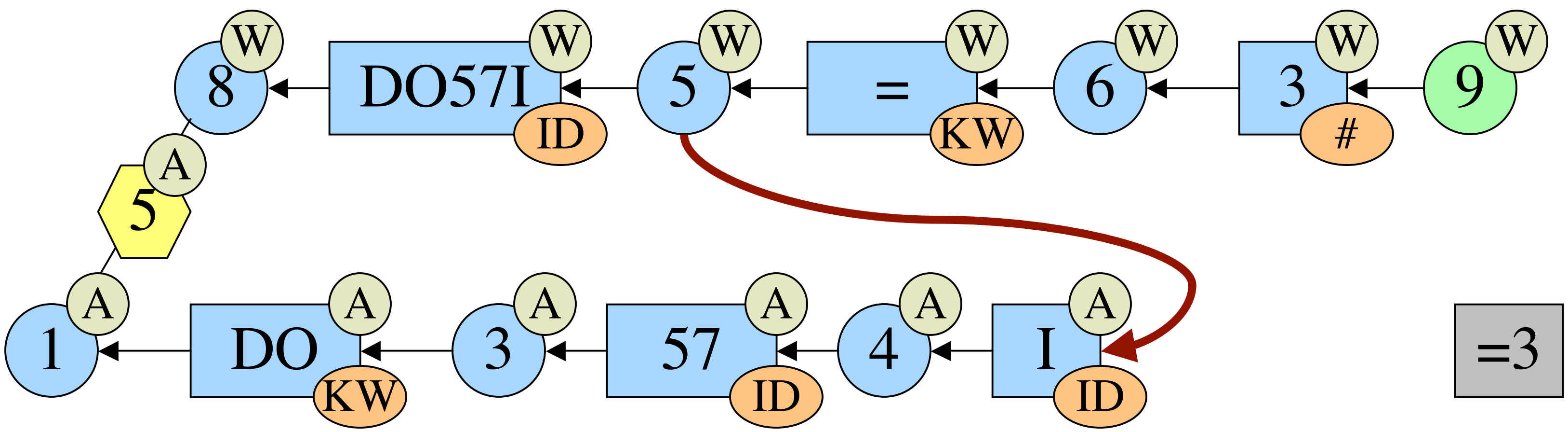
Out of Sync Parsers

- 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*



Implementation

- Keep map: ***lookahead* → *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

HARMONIA

http://harmonia.cs.berkeley.edu