Packrat Parsers Can Support Left Recursion

Alessandro Warth
UCLA / VPRI

James R. Douglass
The Boeing Company

Todd Millstein
UCLA

Packrat Parsing

- Bryan Ford's ICFP'02 "Functional Pearl"
- Memoization of intermediate results → linear parse times
  - Backtracking
  - Unlimited look-ahead
- No ambiguities...

Ordered Choice

- The expression e_1 / e_2 means:
  - try e_1
  - if successful, return result
  - otherwise, backtrack and try e_2
- Makes parser's behavior easy to understand

Packrat Infestation!

- Dozens of implementations
  - Pappy, Rats!, LPeg, ...
- Used in lots of projects
  - Fortress, Matchete, ...
- We use them for program transformation (e.g., OMeta, CAT)

Left Recursion

- Natural way to express syntax of left-associative operators
  - Left recursive rules → left-associative ASTs

```
expr ::= expr “-” number
       / number
```

- Problem: top-down parsers do not support left recursion...

... but packrat parsers are different!

- Intermediate results stored in the parser's **memo table**
- Our paper:

A way to leverage the memo table to support left recursion

Technical Contributions

- Algorithm for supporting left recursion
- Experimental results:
  - typical uses of left recursion supported in linear time
  - very little overhead for non-left-recursive rules
  - can parse heavily left-recursive subset of the Java grammar (as defined in the JLS)

An Alternative Approach

- Rewrite left-recursive grammars
- Technique developed for CFGs, not fully understood in the context of ordered choice
- Pappy [Ford'02] and Rats! [Grimm'06] rewrite directly left-recursive rules
  - Indirectly left-recursive grammars must be rewritten manually

Outline

- Memoization in packrat parsers
- Leveraging memoization to support left recursion
- Further details
- Performance
- Related work

Memoization

<u>Input</u>

1-2-3.

^

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr					
number					
stmt					

Memoization

Input

```
1-2-3.
^
```

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr					
number					
stmt	?				

Memoization

Input

```
1-2-3.
^
```

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr	?				
number					
stmt	?				

Memoization

<u>Input</u>

1-2-3.
 ^

Grammar =

```
{ expr ::= number "-" expr
         / number
  stmt ::= expr ";"
         / expr "." }
```

	0	1	2	3	4
expr	?				
number	1				
stmt	?				

Memoization

<u>Input</u>

1-2-3.
  ^

Grammar =

```
{ expr ::= number "-" expr
         / number
  stmt ::= expr ";"
         / expr "." }
```

	0	1	2	3	4
expr	?				
number	1				
stmt	?				

Memoization

Input

```
1-2-3.
  ^
```

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr	?		?		
number	1				
stmt	?				

Memoization

<u>Input</u>

```
1-2-3.
    ^
```

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr	?		?		
number	1		2		
stmt	?				

Memoization

<u>Input</u>

```
1-2-3.
    ^
```

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr	?		?		
number	1		2		
stmt	?				

Memoization

<u>Input</u>

```
1-2-3.
    ^
```

Grammar =

```
{ expr ::= number "-" expr
         / number
  stmt ::= expr ";"
         / expr "." }
```

	0	1	2	3	4
expr	?		?		?
number	1		2		
stmt	?				

Memoization

<u>Input</u>

```
1-2-3.
     ^
```

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4	
expr	?		?		?	
number	1		2		3	
stmt	?					

Memoization

Input

1-2-3.

^

Grammar =

```
{ expr ::= number "-" expr
         / number
  stmt ::= expr ";"
         / expr "." }
```

	0	1	2	3	4
expr	?		?		?
number	1		2		3
stmt	?				

Memoization

<u>Input</u>

```
1-2-3.
     ^
```

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4	
expr	?		?		?	
number	1		2		3	
stmt	?					

Memoization

Input

1-2-3.
^

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr	?		?		3
number	1		2		3
stmt	?				

Memoization

<u>Input</u>

1-2-3.
^

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr	?		(- 2 3)		3
number	1		2		3
stmt	?				

Memoization

<u>Input</u>

1-2-3.
^

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr	(- 1 (- 2 3))		(- 2 3)		3
number	1		2		3
stmt	?				

Memoization

Input

```
1-2-3.
^
```

Grammar =

```
expr ::= number "-" expr
       / number

stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr	(- 1 (- 2 3))		(- 2 3)		3
number	1		2		3
stmt	?				

Memoization

<u>Input</u>
1-2-3.
^

Grammar = {
```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```
}

	0	1	2	3	4
expr	(- 1 (- 2 3))		(- 2 3)		3
number	1		2		3
stmt	?				

Memoization

<u>Input</u>

```
1-2-3.
      ^
```

Grammar =

```
expr ::= number "-" expr
       / number
stmt ::= expr ";"
       / expr "."
```

	0	1	2	3	4
expr	(- 1 (- 2 3))		(- 2 3)		3
number	1		2		3
stmt	?				

Memoization

Input

1-2-3.

Grammar =

```
expr ::= number “-” expr
       / number
stmt ::= expr “;”
       / expr “.”
```

	0	1	2	3	4
expr	(- 1 (- 2 3))		(- 2 3)		3
number	1		2		3
stmt	(- 1 (- 2 3))				

```
expr ::= number “-” expr
       / number
```

(- 1 (- 2 3))

```
expr ::= number “-” expr
       / number
```

(- 1 (- 2 3))

```
expr ::= expr “-” number
       / number
```

(- (- 1 2) 3)

Left-recursion = trouble

Input
1-2-3.
^

Grammar = {
```
expr ::= expr “-” number
       / number

stmt ::= expr “;”
       / expr “.”
```
}

	0	1	2	3	4
expr					
number					
stmt					

Left-recursion = trouble

Input

1-2-3.
^

Grammar =

```
{ expr ::= expr "-" number
         / number
  stmt ::= expr ";"
         / expr "." }
```

	0	1	2	3	4
expr					
number					
stmt	?				

Left-recursion = trouble

<u>Input</u>

```
1-2-3.
^
```

Grammar =

```
{ expr ::= expr "-" number
         / number
  stmt ::= expr ";"
         / expr "." }
```

	0	1	2	3	4
expr	?				
number					
stmt	?				

Left-recursion = trouble

Input

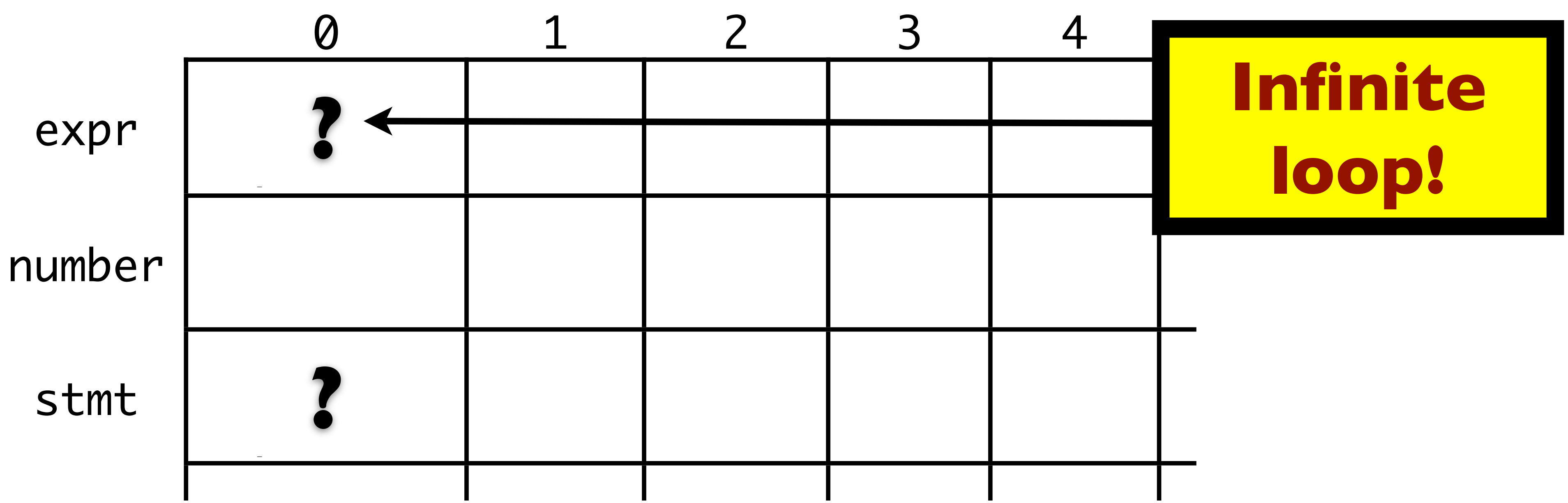
1-2-3.

^

Grammar =

```
expr ::= expr "-" number
       / number

stmt ::= expr ";"
       / expr "."
```

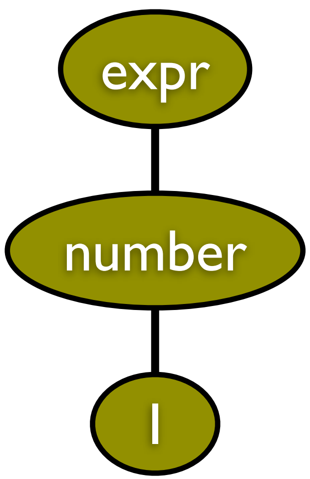


Warning: Super-Duper Important Slide

```
1-2-3                    expr ::= expr "-" number
                                / number
```

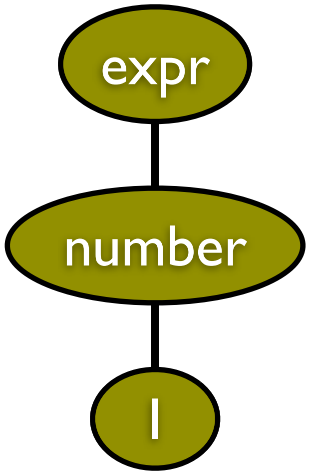
1-2-3

```
expr ::= expr "-" number
       / number
```



1-2-3

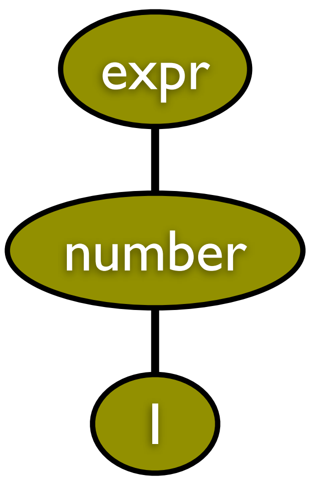
```
expr ::= expr “-” number
       / number
```



seed parse

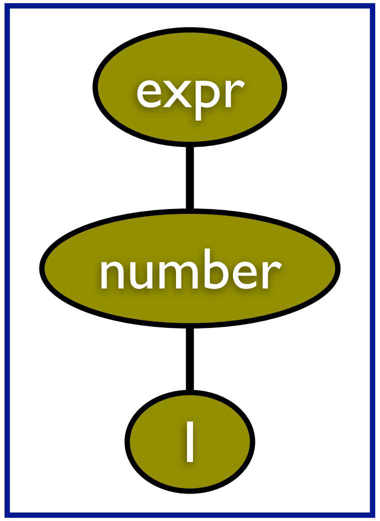
1-2-3

```
expr ::= expr "-" number
       / number
```



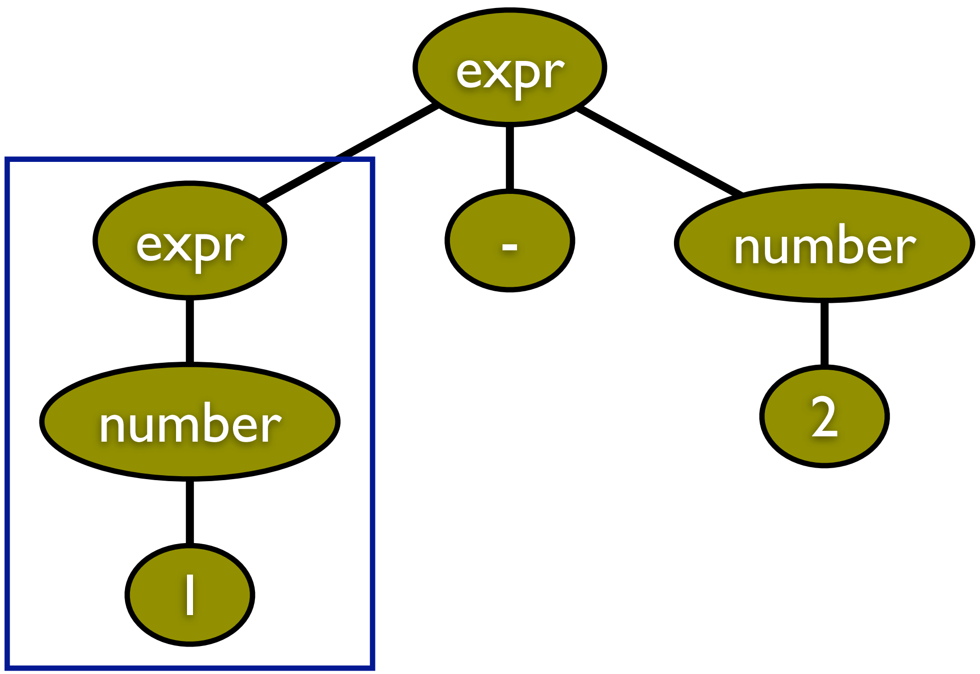
1-2-3

```
expr ::= expr "-" number
       / number
```



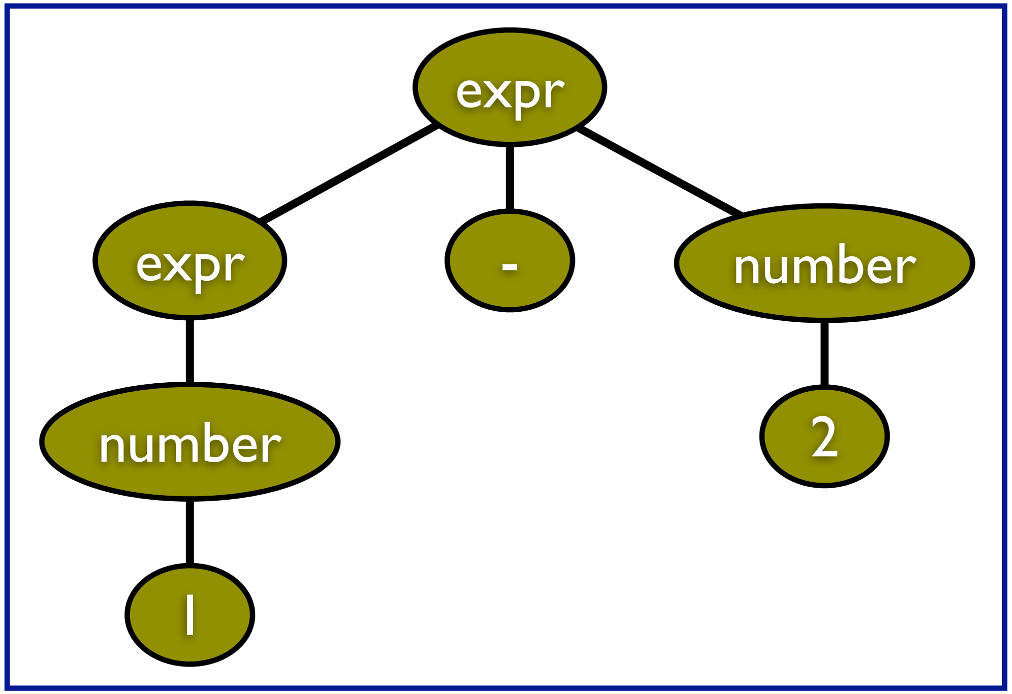
1-2-3

```
expr ::= expr "-" number
       / number
```



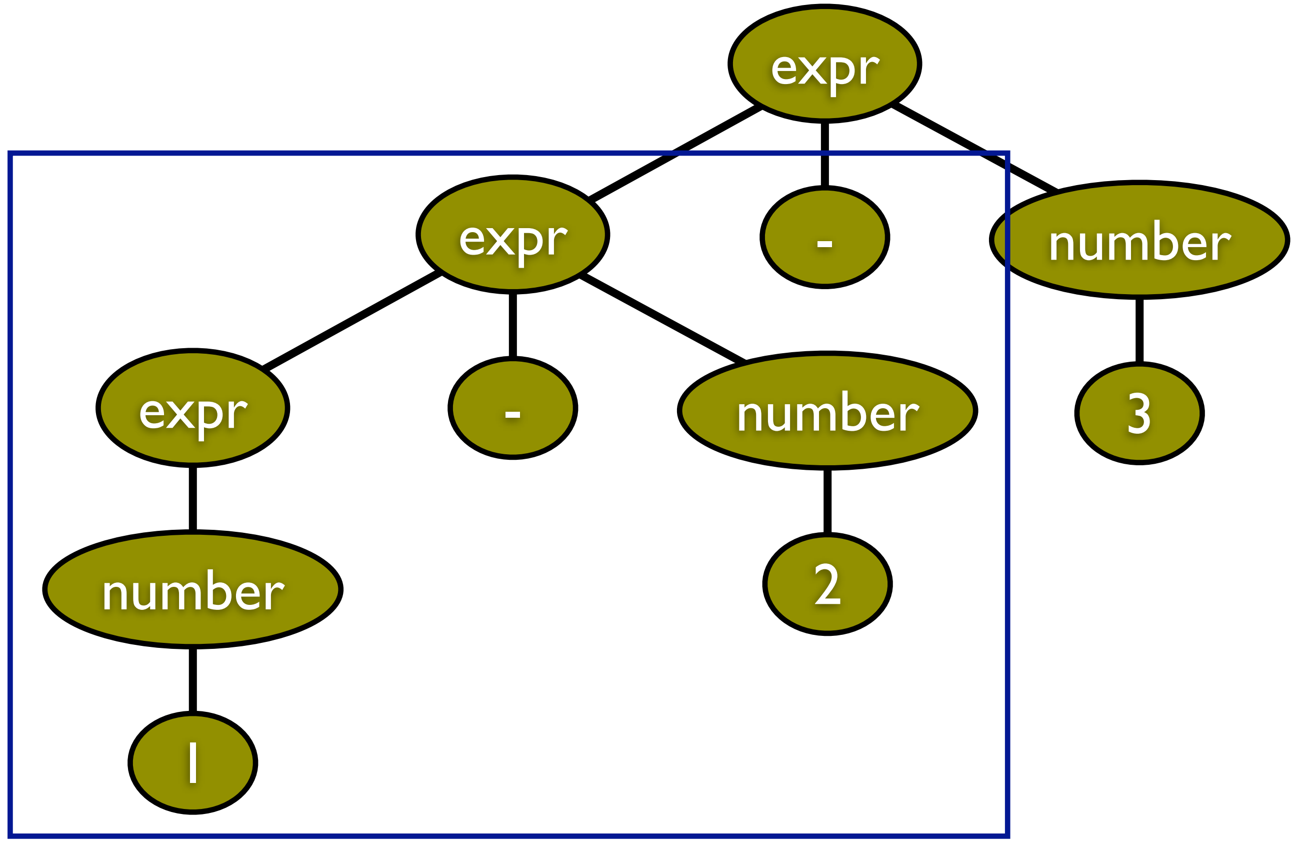
1-2-3

```
expr ::= expr "-" number
       / number
```



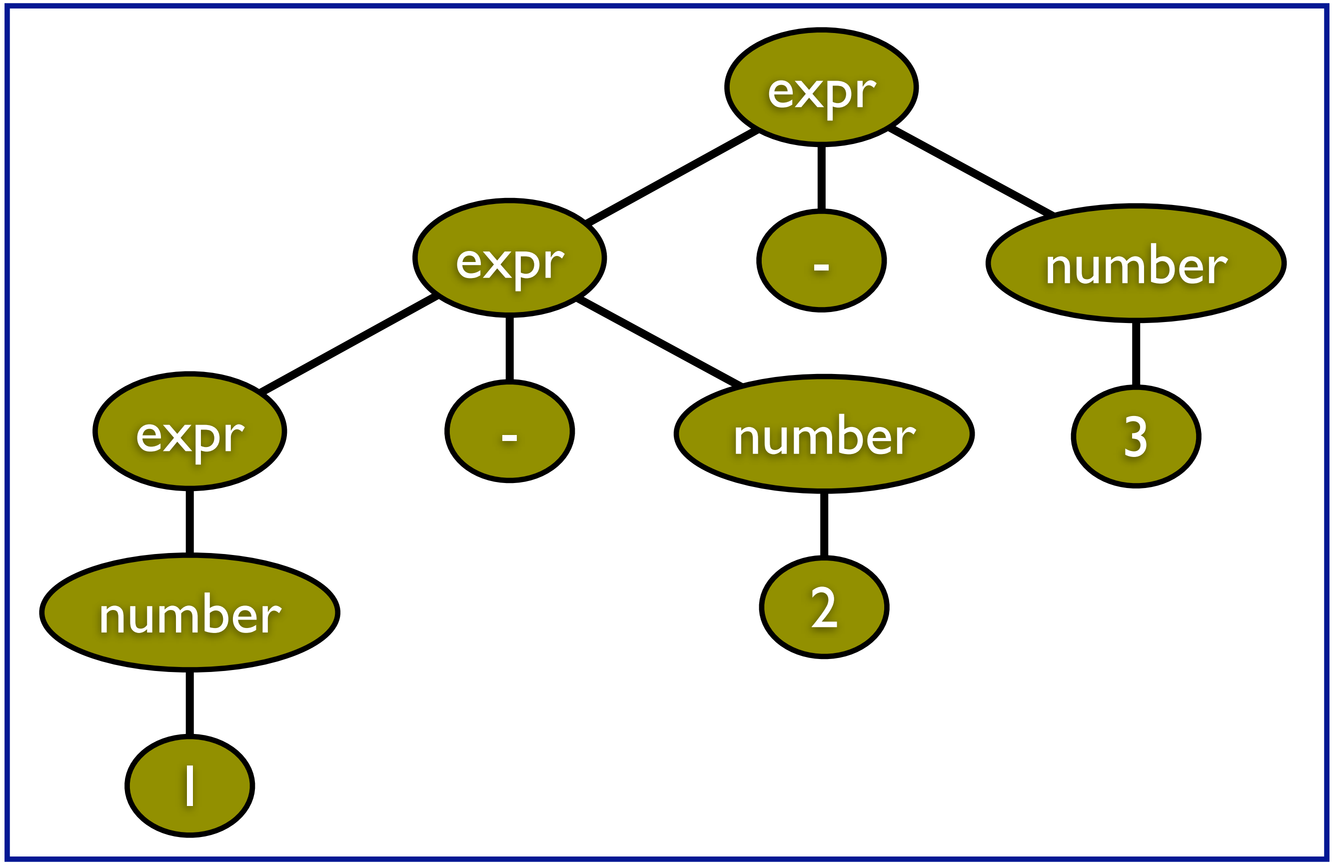
1-2-3

```
expr ::= expr "-" number
       / number
```



1-2-3

```
expr ::= expr "-" number
       / number
```



Finding the Seed

Input = 1-2-3

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr						
number						

Finding the Seed

Input = 1-2-3
　　　 ^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	FAIL					
number						

Finding the Seed

Input = 1-2-3
  ^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	FAIL					
number	1					

Finding the Seed

Input = `1-2-3`
`  ^`

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	1					
number	1					

Finding the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	1					
number	1					

seed parse

Finding the Seed

Input = 1-2-3
 ^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	1					
number	1					

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr “-” number
       / number
```

	0	1	2	3	4	5
expr	1					
number	1					

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	1					
number	1					

Growing the Seed

Input = 1-2-3
    ^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	1					
number	1					

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	1					
number	1					

Growing the Seed

Input = 1-2-3

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	1					
number	1		2			

Growing the Seed

Input = 1-2-3
       ^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	(- 1 2)					
number	1		2			

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	(- 1 2)					
number	1		2			

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	(- 1 2)					
number	1		2			

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	(- 1 2)					
number	1		2			

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	(- 1 2)					
number	1		2		3	

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	(- (- 1 2) 3)					
number	1		2		3	

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	(- (- 1 2) 3)					
number	1		2		3	

Growing the Seed

Input = 1-2-3
^

```
expr ::= expr "-" number
       / number
```

	0	1	2	3	4	5
expr	(- (- 1 2) 3)					
number	1		2		3	

Other Aspects of the Algorithm

- Avoiding unnecessary work for non-left-recursive rules
- Supporting indirect left recursion
- See paper for details

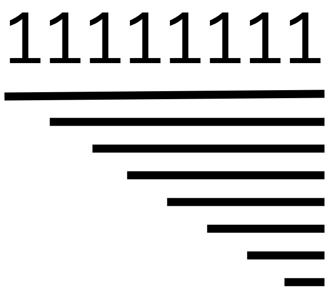
Performance (1)

- Experimental results:
  - Our approach supports typical uses of left recursion in linear time
  - It introduces very little overhead for non-left-recursive rules
  - Left recursion faster than right recursion (w/o tail call optimization)

Performance (2)

- Bad news: possibly super-linear parse times
- Good news: only for contrived grammars

```
ones  ::= ones “1”
        / “1”
start ::= ones “2”
        / “1” start
```



Related Work (1)

- [Frost & Hafiz'06]
  - can support left recursion by limiting otherwise infinite left recursion to `N-1` levels
  - works for any top-down parser, but
    - must know length of input stream
    - $O(n^4)$

Related Work (2)

- [Johnson'95]: technique for building parsers for CFGs
  - based on memoization and CPS
  - left recursion support, polynomial parse times

Related Work (3)

- Katahdin [Seaton'07]
  - language w/ extensible syntax
  - supports rules annotated as left-recursive using similar iterative process
  - does not support indirect left recursion

Conclusion

- Packrat parsers can support left recursion
  - w/o left recursion elimination
  - usually in linear time
  - straightforward implementation
    (see paper)

The End

“Just-in-case” Slides

Ford's Transformation (1)

```
number ::= n:number d:digit → n * 10 + d
         / d:digit          → d
```

↓ ↓

```
number     ::= d:digit f:numberTail → f(d)
numberTail ::= d:digit numberTail:f → λn.f(n * 10 + d)
             / empty                → λx.x
```

Ford's Transformation (2)

Input = `123`

	0	1	2	3
`digit`	1, pos'=1	2, pos'=2	3, pos'=3	**FAIL**
`number`	123, pos'=3	23, pos'=3	3, pos'=3	**FAIL**
`numberTail`			λn.(λx.x)(n * 10 + 3), pos'=3	λx.x, pos'=3

λn.(λn.(λx.x)(n * 10 + 3))(n * 10 + 2),
pos'=3

Ford's Transformation (3)

- From Bryan Ford's thesis:
  - "*As long as the computation of each cell looks up only a limited number of previously-recorded cells in the matrix and completes in constant time, the parsing process as a whole completes in linear time.*"
- At pos=*i*, the function returned by `numberTail` could perform *n-i* additions and multiplications
- So computation of `number` takes O(n)
  - Violates constant time stipulation!!