LL(*): The Foundation Of the ANTLR Parser Generator

Terence Parr
University of San Francisco

Kathleen Fisher
Tufts University

Modern Parsing Strategies

Increased parsing strength through nondeterminism

GLR

- Accepts all grammars
- Based upon bottom-up (*LR*) parsers
- "Forks" subparsers to pursue all possible paths emanating from *LR* states with conflicts
- Merges all successful parses into parse "forest"

PEG/Packrat

- Accepts all non-left-recursive grammars
- Based upon top-down (*LL*) parsers
- Attempt alternatives in order specified; first production to match, wins

Issues With GLR

- Accepts ambiguous grammars (descends from linguistics)
  - **pros**: no pesky conflict warnings, gives all possible interpretations
  - **cons**: no pesky conflict warnings, requires more exhaustive unit testing; subsequent disambiguation phase required
- Ambiguity and actions don't mix: do we exec actions from all successful productions? Which order? Which rule attribute computations to pick?
- Debugging is a challenge
  - Parser state is 1-to-*n* with grammar position
  - Can't use source-level debugger
- Error handling is harder with bottom-up parsers

Issues With PEG

- *PEGs* sometimes do the unexpected:
  $A \rightarrow a \mid ab$
  production *ab* never matches
- Continual speculation and side-effecting actions don’t mix
- Semantic predicates less useful w/o side-effecting user actions
- Debugging: nested backtracking hard to follow
- Can’t detect errors until EOF

Issues with Deterministic LL(k)

- Many natural grammars are not *LL(k)*; also disallows left-recursion
- E.g., no fixed *k* sees past common left prefix *Modifier** in *A* and recursive *E* in *S*:

$$A \rightarrow Modifier^{*}\ Var \mid Modifier^{*}\ Func$$

$$S \rightarrow E\ \mathbf{.} \mid E\ \mathbf{!}$$
$$E \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid i$$

- But, *LL* allows rule parameters, mimics what programmers build by hand, recovers easily, and is easy to debug

Key Insight

- Most parsing decisions don't need backtracking; ~75% are *LL(1)* and ~90% are *LL(k)* for some fixed *k*
- "Infinite" common prefixes are typically only a few tokens
- Can scan past those few tokens with DFA (regex) to see beyond prefix; e.g., can see past prefixes using *Modifier** and **(***i* **)*** for previous grammars
- To minimize speculation, use heterogeneous code generation approach:
  - use *LL(k)* decisions where possible
  - for non-*LL(k)* decisions, try to find a suitable cyclic DFA
  - worst case, rely on backtracking at runtime

LL(*) Strategy

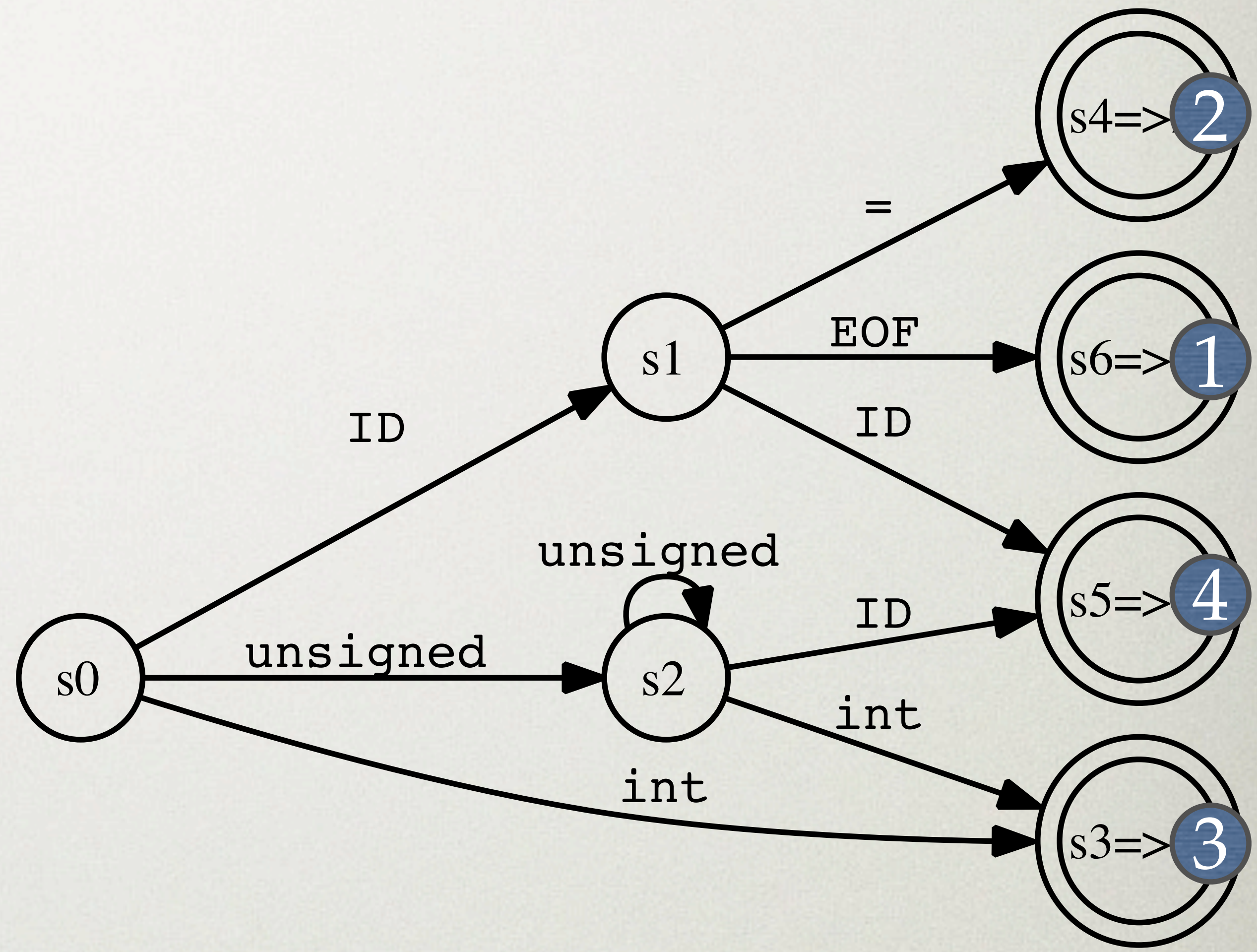
- Statically analyze grammar to construct (potentially cyclic) lookahead DFA per rule (decision)
- Each DFA yields predicted production as function of remaining input
- Graft DFA onto usual *LL* top-down parser to support arbitrary lookahead
- If analysis fails to construct suitable DFA, add backtracking edges to DFA

LL(*) Benefits

- *LL(*)* gracefully throttles up from conventional fixed lookahead to arbitrary lookahead then to backtracking, according to decision complexity
- Even within single DFA, parser dynamically throttles per input sequence: *can* backtrack doesn't imply *will* backtrack (in our samples, from 20% to 74% likelihood per decision)
- No need to specify fixed *k*, discovered automatically (if decidable)
- Analysis detects some ambiguous grammars

Sample grammar / DFA

```
// e.g., x, x=3,
// unsigned unsigned int x
// unsigned T x
s : ID                          1
  | ID '=' expr                 2
  | 'unsigned'* 'int' ID        3
  | 'unsigned'* ID ID           4
  ;
```



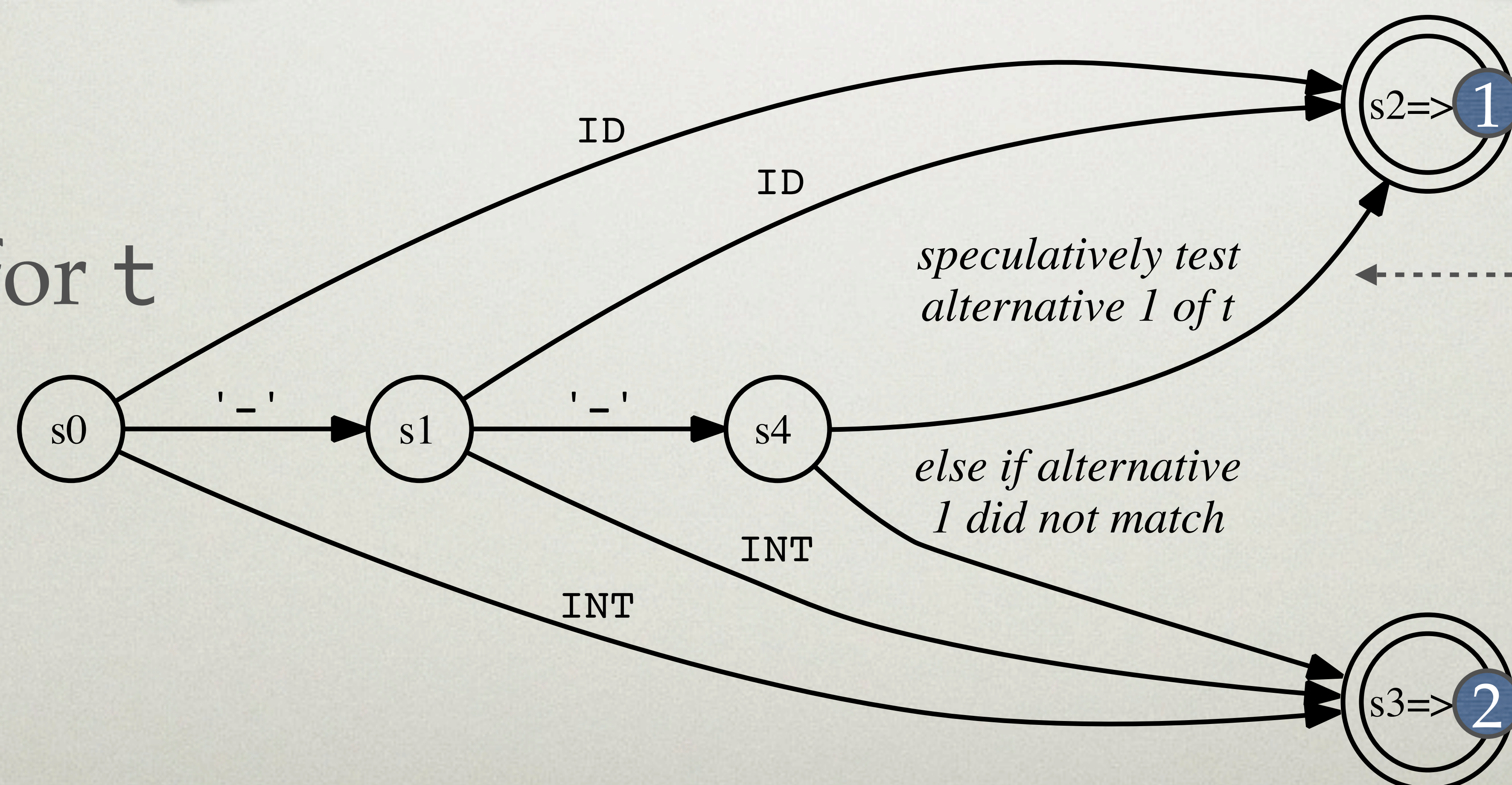
DFA predicts alternatives of rule s; accept states yield production number

Sample Backtracking DFA

```
// e.g., x, -x, --x, 3, -3, --3
options {backtrack=true;}
t   : '-'* ID ‘;’1| expr ‘;’2 ;
expr: INT  | '-' expr ;
```

Says ok to backtrack in DFA via predicated edges

DFA for t



Static LL(*) Analysis, Challenges

- Modified Thompson NFA → DFA subset construction algorithm: ATN → predicated DFA conversion
- Akin to inter-procedural flow analysis; trace graph representation of program, discovering nodes reachable from top-level call site
- Terminals collected along paths emanating from alternative start nodes represent lookahead sequences
- ATN configuration tuple: (ATN state, predicted alternative, ATN call stack, semantic predicate context)
- Existence of suitable *LL(*)* prediction DFA is **undecidable**; must avoid potential infinite loop

Empirical results

Benchmark Grammars

- **Java1.5**: Native ANTLR grammar; PEG mode[1]
- **RatsC, RatsJava**: *Rats!* grammars manually converted to ANTLR syntax using PEG mode while preserving the essential structure.
- **VB.NET, TSQL, C#**: Commercial grammars for Microsoft languages provided by *Temporal Wave, LLC*

[1]Fail over to backtracking when needed

Static Analysis Results

Grammar	Lines	n	Fixed	Cyclic	Backtrack	Runtime
Java1.5	1,022	170	150	1	20 (11.8%)	3.1s
RatsC	1,174	143	111	0	32 (22.4%)	2.8s
RatsJava	763	87	73	6	8 (9.2%)	3s
VB.NET	3,505	348	332	0	16 (4.6%)	6.75s
TSQL	8,241	1,120	1,053	10	57 (5.1%)	13.1s
C#	3,476	217	189	2	26 (12%)	6.3s

- **RatsJava** grammar designed as backtracking grammar, but ANTLR removes all but 9% of backtracking
- Most backtracking removed even for big grammars such as TSQL (8241 line grammar needs backtracking in only 5% of its decisions)

Most Decisions Are LL(k)

Grammar	$LL(k)$	$LL(1)$	Lookahead depth k					
			1	2	3	4	5	6
Java1.5	88.24%	74.71%	127	20	2	1		
RatsC	77.62%	72.03%	103	7	1			
RatsJava	83.91%	73.56%	64	8	1			
VB.NET	95.40%	88.79%	309	18	4	1		
TSQL	94.02%	83.48%	935	78	11	14	9	6
C#	87.10%	78.34%	170	19				

- ANTLR automatically detects *k*; user does not need to specify
- Note commercial grammars likely reorganized to reduce arbitrary lookahead requirements (as optimization); e.g., TSQL is 94% *LL(k)*

Parse-Time results

Grammar	Input lines	parse-time	n	avg k	back. k	max k
Java1.5	12,416	78ms	111	1.09	3.95	114
RatsC	37,019	771ms	131	1.88	5.87	7,968
RatsJava	12,416	412ms	78	1.85	5.95	1,313
VB.NET	4,649	351ms	166	1.07	3.25	12
TSQL	794	13ms	309	1.08	2.63	20
C#	3,807	524ms	146	1.04	1.60	9

ouch!

- On average, parsers look just 1 to 2 tokens ahead
- When backtracking, parsers look < 6 tokens ahead
- Max lookahead for backtracking can be huge, though; can reorganize to optimize

Summary

- Nondeterminism provides parsing power but can allow undesirable grammar constructs and negatively impacts debugging, error handling, and arbitrary action support
- *LL(*)* mostly deterministic but as expressive as PEGs and beyond due to predicates
- *LL(*)* construction is undecidable, but our algorithm detects potential infinite loops and fails over to backtracking, throttling from fixed to arbitrary lookahead to backtracking
- Experiments and widespread use indicates *LL(*)* hits a sweet spot in the parsing spectrum (134,576 downloads January 9, 2008 - October 28, 2010 per Google Analytics)

Work Related to LL(*)

- *LL(*)* inspired by *LL*-regular grammars (Jarzabek, Krawczyk, and Nijholt in 1970s); they provided linear two-pass parser
- We give one-pass parser and extend to support semantic predicates and graceful backtracking fail over
- Analogous to *LR*-regular inspired *LAR(m)* parsers (Bermudez & Schimpf)
- *LL(*)* is optimization of PEG as GLR is optimization of Earley's algorithm