Better Extensibility through Modular Syntax

Robert Grimm
New York University

Syntax Matters

- More complex syntactic specifications
  - Extensions to existing programming languages
    - Transactions, event-based code, network protocols, information flow, device drivers,…
  - Several dialects of same programming language
    - Think K&R, ISO, and GCC for C
  - Distinct languages with considerable syntactic overlap
    - Think C, C++, Objective-C, Java, C#
- More programming language tools
  - Compilers, interpreters, syntax-highlighting editors, API documentation generators, source measurement tools

Syntax Matters

- More complex syntactic specifications
  - Extensions to existing programming languages
    - Transactions, event-based code, network protocols, information flow, device drivers,…
  - Several dialects of same programming language
    - Think K&R, ISO, and GCC for C
  - Distinct languages with considerable syntactic overlap
    - Think C, C++, Objective-C, Java, C#
- More programming language tools
  - Compilers, interpreters, syntax-highlighting editors, API documentation generators, source measurement tools

Need easily extensible syntax and parsers!

Three Desirable Properties

- Suitable formalism
  - Closed under composition to enable modularity
  - Unambiguous as computer formats have one meaning
  - Scannerless to also provide extensibility at lexical level
- Expressive module system
  - Units of productions to provide encapsulation
  - Modifications of units to capture extensions and subsets
  - Flexible composition of units to maximize reuse
- Well-defined escape hatch
  - No formalism can capture all languages

CFGs Fall Short

- LR, LL parsing
  - LALR used by Yacc; LL used by ANTLR, JavaCC
  - But not closed under composition
- GLR, Earley, CYK parsing
  - GLR used by Bison, Elkhound, SDF2
  - Closed under composition
  - But not unambiguous
    - Building all possible trees is inefficient
    - Heuristically selecting one tree may result in wrong one
    - Requiring explicit disambiguation adds complexity

PEGs Are More Suitable

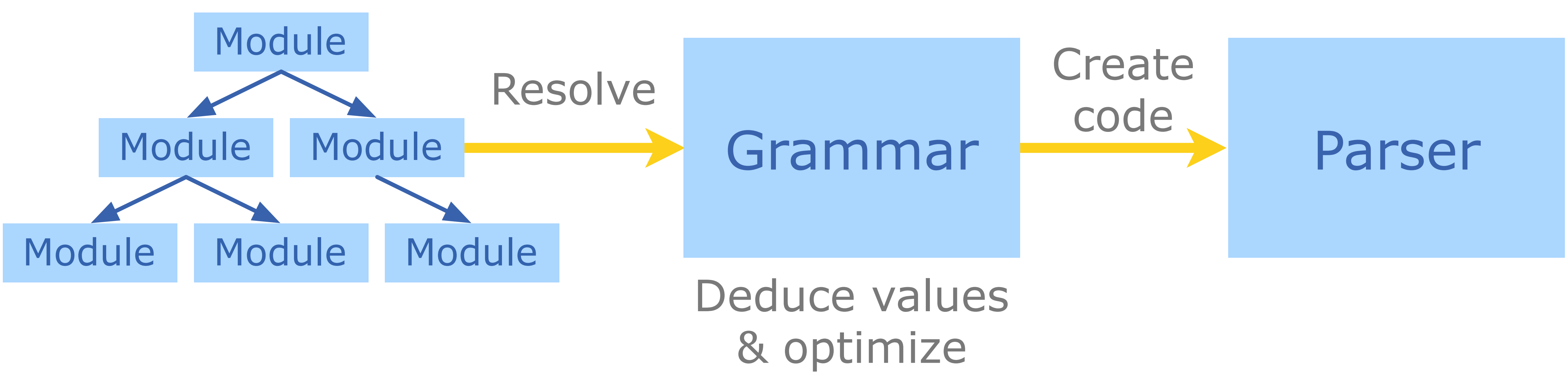
- Parsing expression grammars (PEGs)
  - Basic theory introduced by Birman [PhD '70]
  - Fully developed by Ford [ICFP '02, POPL '04]
  - Closed under composition, intersection, complement
  - Ordered choices to avoid ambiguities
  - Syntactic predicates to increase expressiveness [Parr '94]
    - Match but do not consume input
  - Scannerless to avoid separate lexer
  - Implemented by recursive descent parsers
    - Memoize all results to ensure linear time performance

PEGs Are More Suitable

- Parsing expression grammars (PEGs)
  - Basic theory introduced by Birman [PhD '70]
  - Fully developed by Ford [ICFP '02, POPL '04]
  - Closed under composition, intersection, complement
  - Ordered choices to avoid ambiguities
  - Syntactic predicates to increase expressiveness [Parr '94]
    - Match but do not consume input
  - Scannerless to avoid separate lexer
  - Implemented by functional "packrat parsers"
    - Memoize all results to ensure linear time performance

My Work

- *Rats!*, a packrat parser generator for Java
  - Makes PEGs practical for imperative languages
    - Concise syntax, aggressive optimizations
  - Provides expressive module system
  - Supports global state through lightweight transactions



Talk Outline

- Introduction
- Module system
- Parser implementation and optimizations
- Experimental evaluation
- Conclusions

Productions

- Basic format
  - *Attribute* Type Nonterminal = Expression ;*
- Operators
  - EBNF-like notation
    - Literals; sequences; greedy choices ('/'), repetitions, options
  - Syntactic predicates
    - Followed-by ('&'), not-followed-by ('!')
  - Support for semantic values
    - Actions ("{…}"), bindings ("id:*e*"), predicates ("&{…}")
    - Not necessary when returning null, strings, generic tree nodes, when passing value through

Modules

- Provide encapsulation
  - Group related productions
  - Track dependencies

```
module xtc.util.Symbol;
import xtc.util.Spacing;

String Symbol = SymbolCharacters Spacing ;

transient String SymbolCharacters =
    <GreaterGreaterEqual> ">>="
  / <LessLessEqual>       "<<="
  / <GreaterGreater>      ">>"
  / <LessLess>            "<<"
    /* and so on ... */ ;
```

Visibility Control

- Productions declare visibility through attribute
  - "public" = top-level production, visible to outside
  - "protected" = inter-module production
  - "private" = intra-module, helper production
- For ambiguous nonterminals
  - Give precedence to productions defined in same module
  - If all productions in other modules, require qualified name
    - E.g., "xtc.util.Spacing.Spacing" instead of "Spacing"

Visibility Control

- Productions declare visibility through attribute
  - "public" = top-level production, visible to outside
  - "protected" = inter-module production, default
  - "private" = intra-module, helper production
- For ambiguous nonterminals
  - Give precedence to productions defined in same module
  - If all productions in other modules, require qualified name
    - E.g., "xtc.util.Spacing.Spacing" instead of "Spacing"

(Back to) Modules

- Provide encapsulation
  - Group related productions
  - Specify dependencies

```
module xtc.util.Symbol;
import xtc.util.Spacing;

String Symbol = SymbolCharacters Spacing ;

transient String SymbolCharacters =
    <GreaterGreaterEqual> ">>="
  / <LessLessEqual>       "<<="
  / <GreaterGreater>      ">>"
  / <LessLess>            "<<"
    /* and so on ... */ ;
```

Module Parameters

- Facilitate flexible composition
  - Represent module names, are replaced on instantiation
  - Delay provision of actual names until parser generation time

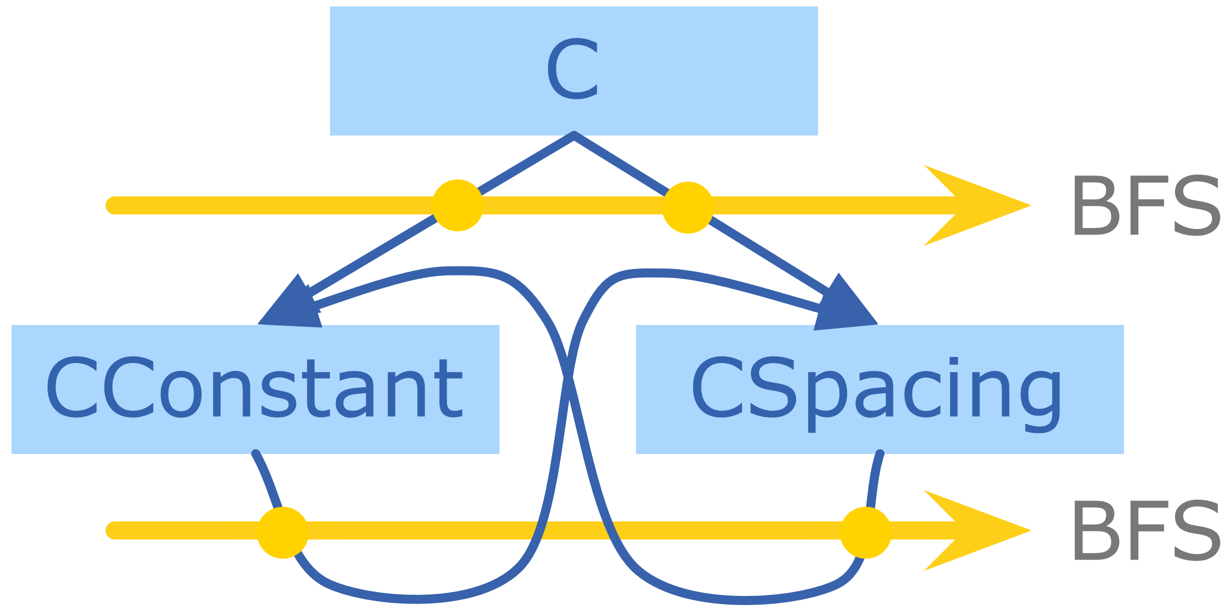
```
module xtc.util.Symbol(Spacing);
import Spacing;

String Symbol = SymbolCharacters Spacing ;

transient String SymbolCharacters =
    <GreaterGreaterEqual> ">>="
  / <LessLessEqual>       "<<="
  / <GreaterGreater>      ">>"
  / <LessLess>            "<<"
    /* and so on ... */ ;
```

Module Resolution

- Breadth-first search across dependency declarations
  - Includes explicit instantiations of parameterized modules

```
instantiate xtc.lang.CConstant(xtc.lang.CSpacing);
instantiate xtc.lang.CSpacing(xtc.lang.CState,
                              xtc.lang.CConstant);
```

  - Supports circular dependencies



  - Best practice: Parameterize all modules, instantiate at top

Module Modifications

- Concisely express how modules differ from another
  - Can add, override, or remove individual alternatives
    - Can also override entire productions, incl. attributes
  - Result in new modules, combining deltas and bases

```
module xtc.lang.JavaSymbol(Symbol);
modify Symbol;

String SymbolCharacters +=
    <TripleGreaterEqual>  ">>>="
  / <GreaterGreaterEqual> ... ;
String SymbolCharacters +=
    <TripleGreater>  ">>>"
  / <GreaterGreater> ... ;
```

Putting It All Together

- Three modules

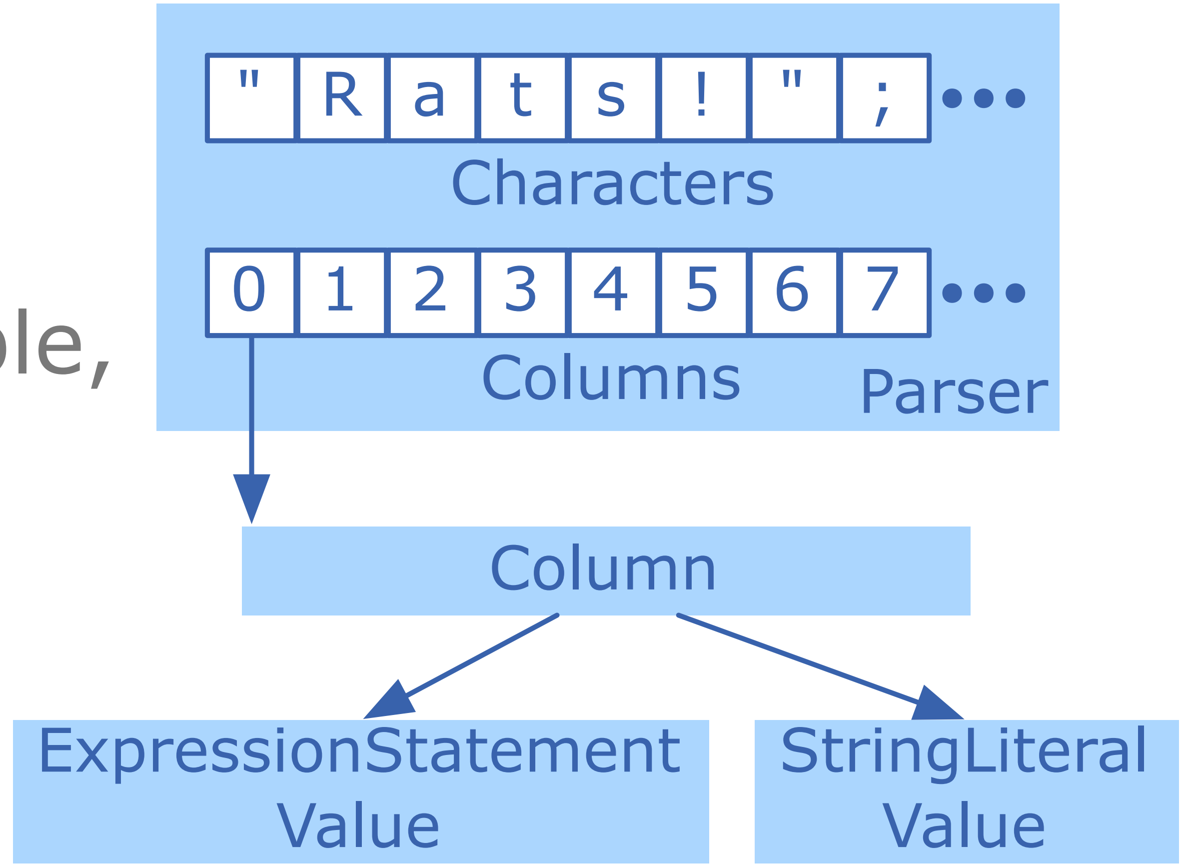
xtc.util.Symbol	Symbols common to C and Java
xtc.lang.JavaSymbol	Symbols unique to Java
xtc.lang.CSymbol	Symbols unique to C

- Considerable flexibility

JavaSymbol modifies Symbol	All of Java's symbols
CSymbol modifies Symbol	All of C's symbols
JavaSymbol modifies CSymbol modifies Symbol	All of Java's and C's symbols

Parser Implementation

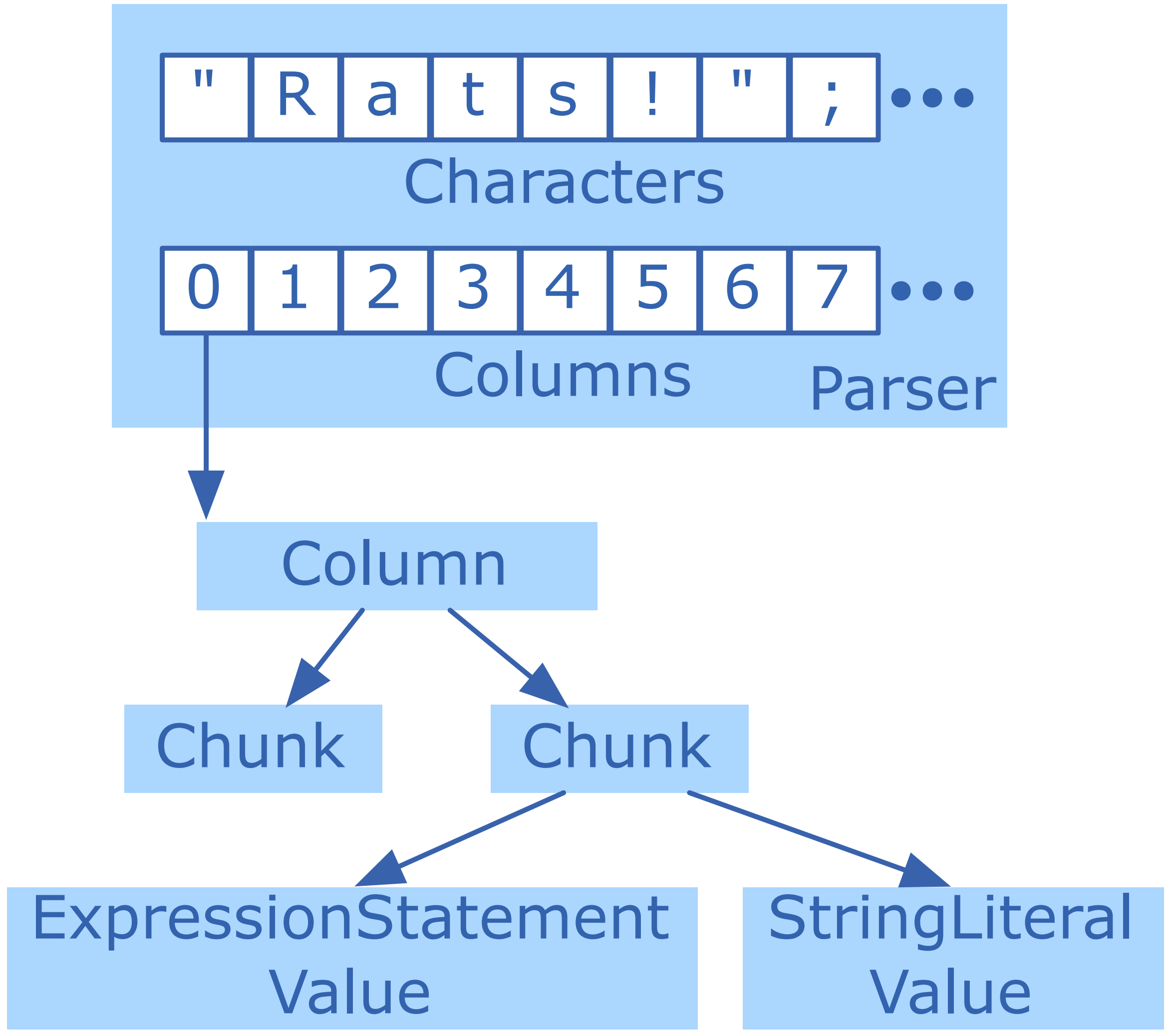
- Method for each production
- Character array for input
- Column array for memo table, with field per production
  - Null implies not tried before
  - SemanticValue represents successful parse
    - <actual value, index of next column, possible parse error>
  - ParseError represents failed parse
    - <error message, index of error>
  - Result provides common interface to values and errors



Avoid Empty Fields

[Ford '02]

- Insight: Most productions not tried for input position
  - Field remains null
- Break column object into chunks
  - Allocate only when needed
    - I.e., when one of chunk's productions is tried



Avoid Memoization

- Insight: Most productions tried 0-1× for input position
  - Token-level: Most helper productions, spacing
  - Hierarchical syntax: Look at tokens before references
    - If different, production can only be tried at most once
- Give grammar writers control over memoization
  - "transient" attribute disables memoization
    - Doubly effective: Eliminates rows & columns from memo table
    - Facilitates further optimizations
      - Preserve repetitions in transient productions as iterations
      - Turn direct left-recursions into equivalent right-iterations

Improve Terminal Recognition

- Insight: Many alternatives in token-level productions start with different characters
  - Inline sole nonterminals (if productions are transient)
  - Combine common prefixes
  - Use switch statements for disjoint alternatives
- Also: Avoid dynamic instantiation of matched text
  - Use string if text can be statically determined
  - Use null if text is never used (i.e., bound)

Suppress Unnecessary Results

- Insight: Many productions pass the value through
  - Example: 17 levels of expressions for C or Java, all of which must be invoked to parse a literal, identifier,…
  - Only create new SemanticValue if contents differ
    - Otherwise, reuse passed-through value
- Insight: Most alternatives fail on first expression
  - Example: Statement production for C or Java
  - Only create new ParseError if subsequent expressions or entire production fail
    - Meanwhile, use generic error object

Experimental Evaluation

- Syntactic specification
  - Are grammars concise?
  - Are grammars extensible?
- Parser performance
  - What is optimizations' impact?
  - Are parsers fast enough?

Experimental Evaluation

- Syntactic specification
  - Are grammars concise? Yes, see paper!
  - Are grammars extensible?
- Parser performance
  - What is optimizations' impact?
  - Are parsers fast enough?

Experimental Setup

- Five contestants, using Java 1.4 grammars

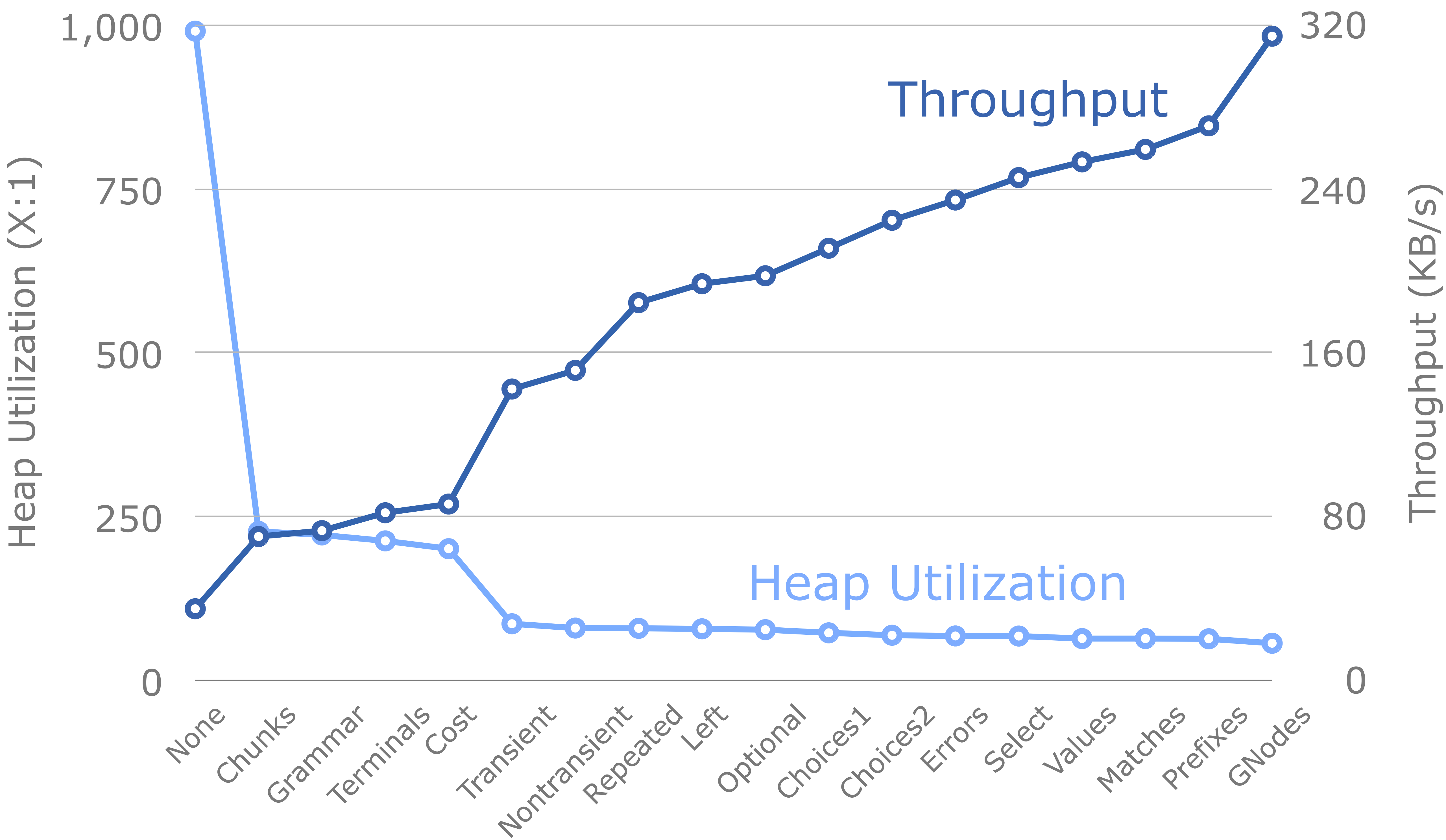
	Formalism	Language	Version
Rats!	PEG	Java	1.8.0
SDF2	GLR	C	2.3.3
Elkhound	LALR/GLR	C++	2005.08.22b
ANTLR	LL	Java	2.7.5
JavaCC	LL	Java	4.0

- 41 source files: 1-135 KB, varying programming styles
- 1 judge: Apple iMac from fall '02
  - Reporting least-squares-fit

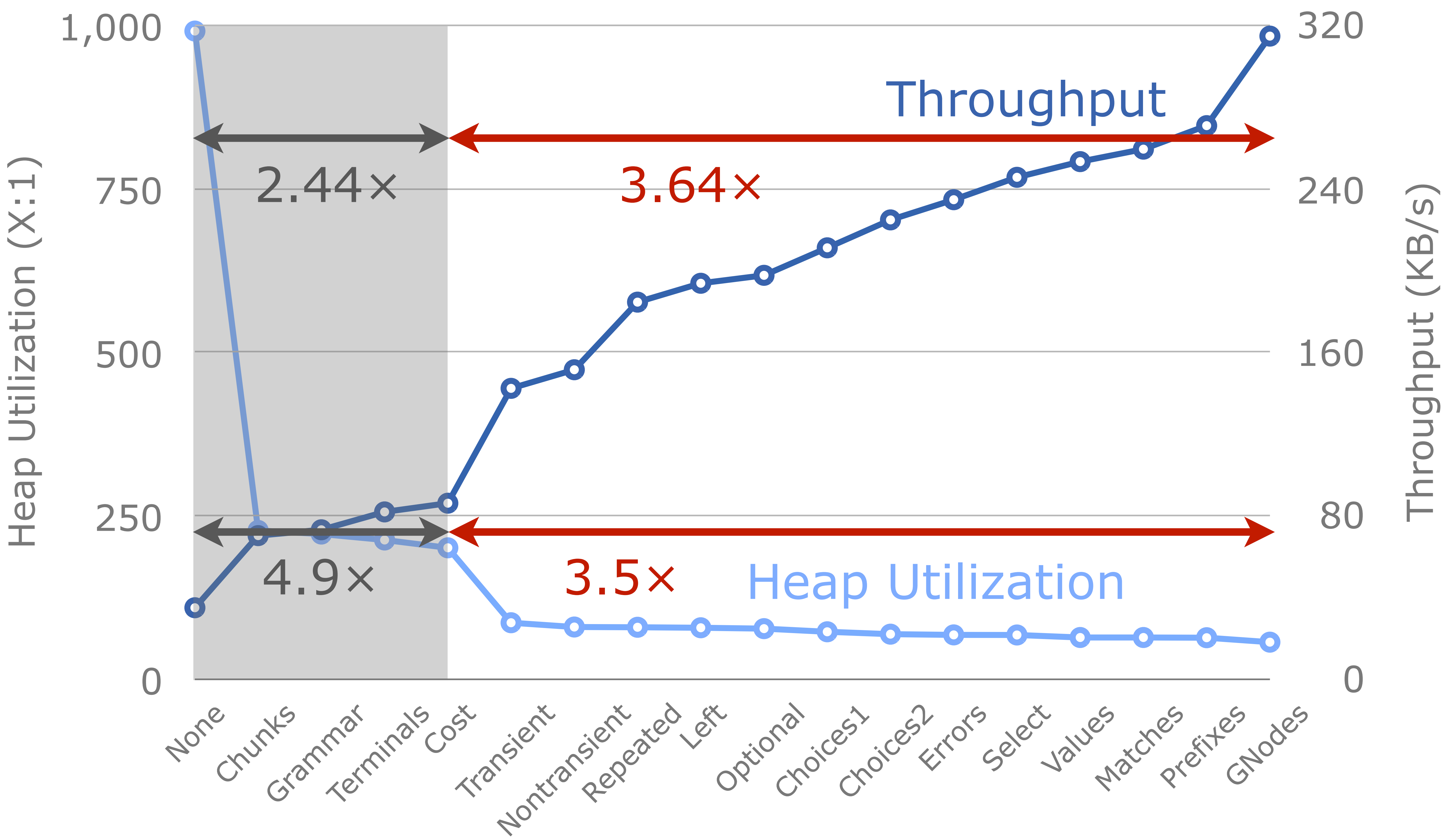
Grammars Are Easily Extensible

- C4 (CrossCutting C Compiler)
  - Aspect-enhanced C to simplify Linux kernel extensions
  - 17 hours (4 learning, 13 writing & debugging)
  - 4 modules with 150 LoC + 1 Java class with 130 LoC
- Jeannie
  - Combination of Java and C to simplify JNI programming
    - Just write: `` jobject result = `obj.method(…); ``
  - 45 hours (25 learning, 20 writing & debugging)
  - 4 modules with 230 LoC

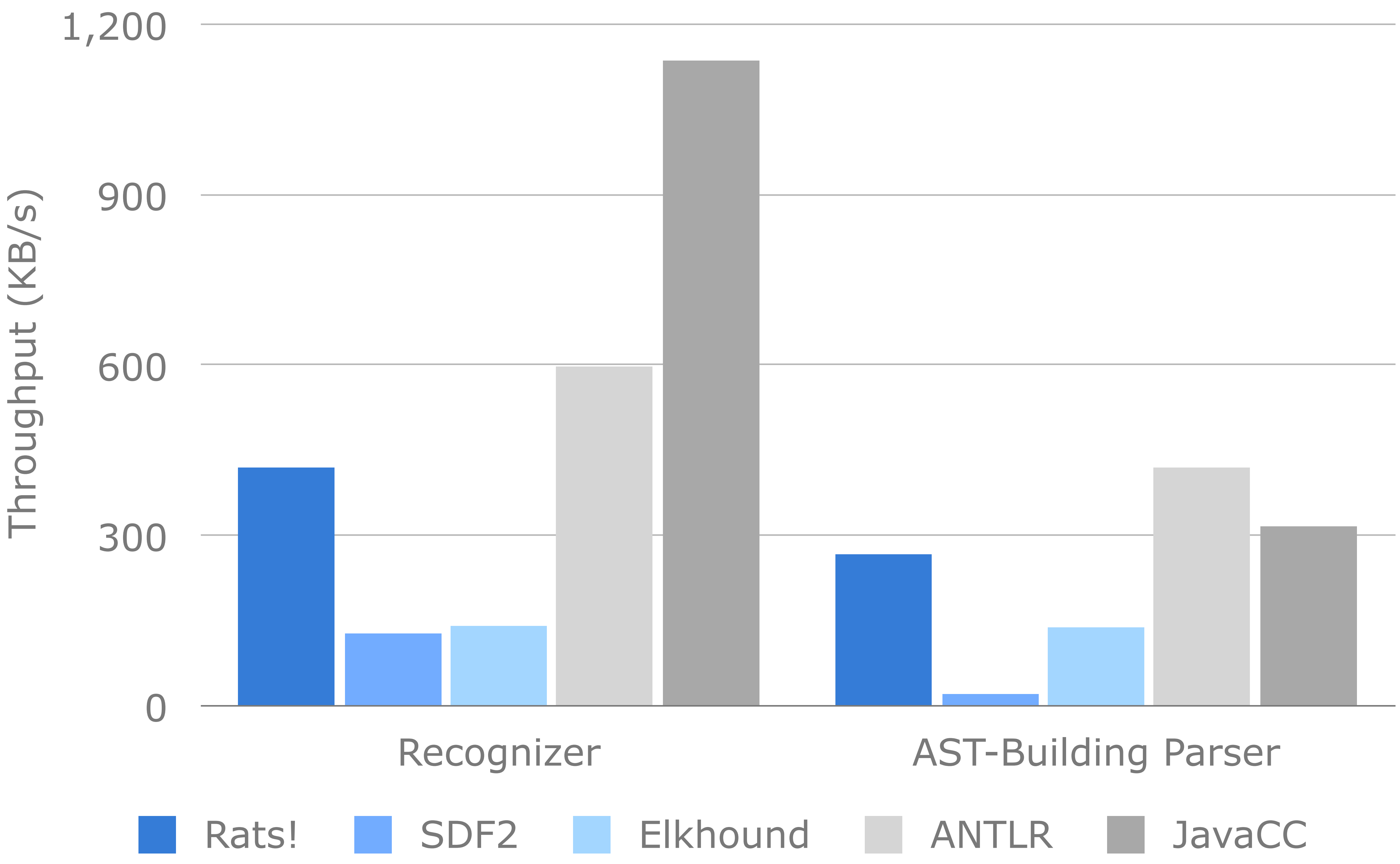
Optimizations Are Effective



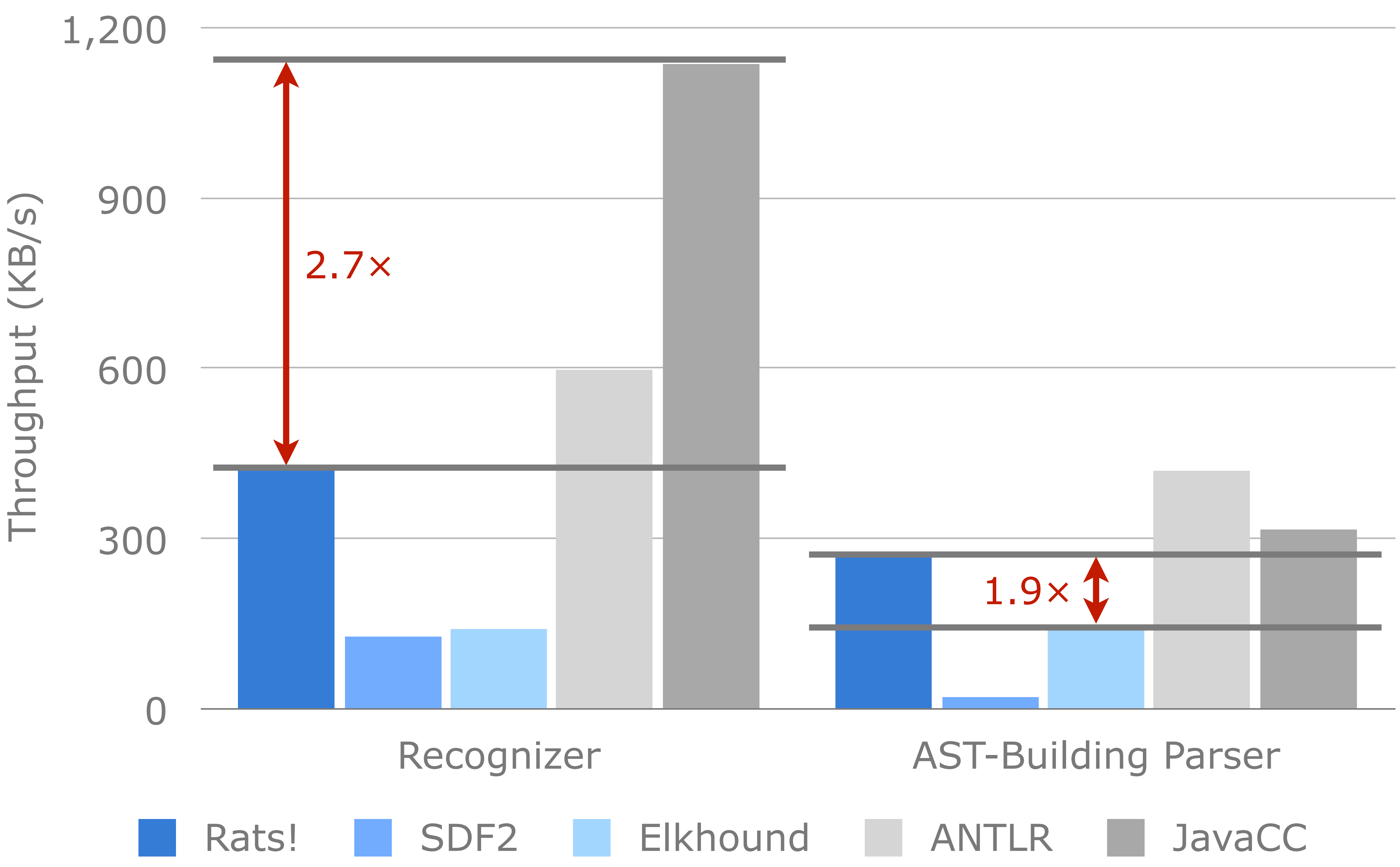
Optimizations Are Effective



Parsers Perform Well



Parsers Perform Well



Conclusions

- Made PEGs practical for imperative languages
  - Concise syntax, global state, aggressive optimizations
- Built an expressive module system
  - Modules to encapsulate related productions
  - Modifications to concisely specify extensions and subsets
  - Parameters to enable flexible composition and reuse
- To good overall effect
  - Others can realize real-world extensions in little time, code
  - Parsers perform reasonably well

http://cs.nyu.edu/rgrimm/xtc/

Thank you: Martin Bravenboer,
Marc Fiuczynski, Bryan Ford,
Ben Goldberg, Laune Harris,
Martin Hirzel, Trevor Jim,
Scott McPeak, Marco Yuen