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

![Diagram showing the process from modules to grammar to parser]

- Resolve
- Deduce values & optimize
- Create code
- Parser
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
* Provide encapsulation
  * Group related productions
  * Track dependencies

```java
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"
* Provide encapsulation
  * Group related productions
  * Specify dependencies

```java
module xtc.util.Symbol;
import xtc.util.Spacing;
String Symbol = SymbolCharacters Spacing ;
transient String SymbolCharacters =
    <GreaterGreaterGreaterEqual> ">>=
    / <LessLessLessEqual>      "<<="
    / <GreaterGreaterGreater>  ">>"
    / <LessLessLess>           "<<"
/* and so on ... */ ;
```
Facilitate flexible composition

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

```java
module xtc.util.Symbol(Spacing);
import Spacing;
String Symbol = SymbolCharacters Spacing ;
transient String SymbolCharacters =
    <GreaterGreaterGreaterEqual> ">>="
    / <LessLessEqual>       "<<<<"
    / <GreaterGreater>      ">>"
    / <LessLess>            "<<"
    /* and so on ... */ ;
```
Module Resolution

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

```plaintext
module xtc.lang.JavaSymbol(Symbol);
modify Symbol;
String SymbolCharacters +=
    <TripleGreaterEqual> ">>>="
/ <GreaterGreaterEqual> ... ;
String SymbolCharacters +=
    <TripleGreater> ">>>"
/ <GreaterGreater> ... ;
```
Putting It All Together

- Three modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xtc.util.Symbol</td>
<td>Symbols common to C and Java</td>
</tr>
<tr>
<td>xtc.lang.JavaSymbol</td>
<td>Symbols unique to Java</td>
</tr>
<tr>
<td>xtc.lang.CSymbol</td>
<td>Symbols unique to C</td>
</tr>
</tbody>
</table>

- Considerable flexibility

<table>
<thead>
<tr>
<th>Modification</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaSymbol modifies Symbol</td>
<td>All of Java's symbols</td>
</tr>
<tr>
<td>CSymbol modifies Symbol</td>
<td>All of C's symbols</td>
</tr>
<tr>
<td>JavaSymbol modifies CSymbol</td>
<td>All of Java's and C's symbols</td>
</tr>
</tbody>
</table>
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

- 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

[Ford '02]
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
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)
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

<table>
<thead>
<tr>
<th>Contestant</th>
<th>Formalism</th>
<th>Language</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Rats!</em></td>
<td>PEG</td>
<td>Java</td>
<td>1.8.0</td>
</tr>
<tr>
<td>SDF2</td>
<td>GLR</td>
<td>C</td>
<td>2.3.3</td>
</tr>
<tr>
<td>Elkhound</td>
<td>LALR/GLR</td>
<td>C++</td>
<td>2005.08.22b</td>
</tr>
<tr>
<td>ANTLR</td>
<td>LL</td>
<td>Java</td>
<td>2.7.5</td>
</tr>
<tr>
<td>JavaCC</td>
<td>LL</td>
<td>Java</td>
<td>4.0</td>
</tr>
</tbody>
</table>

- 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

Heap Utilization (X:1) vs. Throughput (KB/s)

- None
- Chunks
- Grammar
- Terminals
- Cost
- Transient
- Nontransient
- Repeated
- Left
- Optional
- Choices1
- Choices2
- Errors
- Select
- Values
- Matches
- Prefixes
- GNodes

Throughput:
- None: 0 KB/s
- Chunks: 320 KB/s
Optimizations Are Effective

Heap Utilization (X:1)

Throughput (KB/s)

Throughput

Heap Utilization

Optimizations Are Effective

Throughput 3.64×

3.5×

2.44×
Parsers Perform Well

Throughput (KB/s)

Recognizer

AST-Building Parser

- Rats!
- SDF2
- Elkhound
- ANTLR
- JavaCC
Parsers Perform Well

Throughput (KB/s)

Recognizer

- Rats!
- SDF2
- Elkhound

AST-Building Parser

- ANTLR
- JavaCC

Throughput Comparison:

- Rats! (2.7×)
- SDF2
- Elkhound
- ANTLR (1.9×)

Graph showing performance of different parsers.
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