Towards an eXTensible Compiler
Step 1: The Parser

Robert Grimm
New York University
Motivation

- Wide-spread need for extensible languages and compilers
  - Systems researchers
    - libasync to build asynchronous, distributed applications
    - Capriccio to build scalable, multi-threaded servers
    - MACEDON to build overlay networks
  - Systems builders
    - PHP, JSP, ASP.NET
  - Language researchers
    - Cyclone, CCured, MultiJava, HydroJ
  - Teachers
    - DrScheme
Existing solutions are not very attractive (especially to systems researchers/builders)

- C preprocessor has limited expressive power, is unsafe
- C++ templates are Turing-complete (!) but also complex and still not enough
- Extending the compiler is hard
  - GCC is a very complex beast
  - CIL [Necula et al. ‘02] and Polyglot [Nystrom et al. ‘03] are targeted at compiler experts
Enter xtc

- Macro system for C and a corresponding compiler
  - Expressive
    - Supports module system, objects, generics
  - Safe
    - Is not only hygienic, but also type-safe
  - Efficient
    - Maintains semantic information throughout the compiler
    - Supports user-specified optimizations
A toolkit for building extensible source-to-source transformers
  - Initially, written in and targeted at Java
    - Simple object system, GC, collections (lists, hashmaps)
  - Later, ported over to xtc itself
  - Leave conventional optimizations and codegen to javac/gcc

A set of interrelated domain-specific languages
  - Support syntax specification and desugaring, typing constraints, optimizations
  - Implemented on top of the toolkit

Inspired by Metal [Hallem et al. ‘02]
Why C?

- Still is a popular language for implementing operating and distributed systems
- Unlike Java or C#, does not require a virtual machine
  - Which, ironically, is often implemented in C
- Unlike C++, Objective-C, Java, or C#, has no object model
  - C++ objects provide everything and the kitchen sink
  - Java’s object model is simple, but has performance issues
Talk Outline

- Motivation and vision for xtc
- Introduction to *Rats!*, our parser generator
- Optimizations
- Evaluation
- Our compiler framework so far
- Future work and conclusions
Why yet another parser generator?

- Grammar provides best hooks to changing a language
  - Unless the syntax is highly regular – think Scheme

Goals

- Easy to specify and extend grammars
  - Including adding, changing, removing productions
- Reasonable performance
Basic strategy: Packrat parsing
- Originally described by Birman [Phd ‘70]
- Rediscovered by Ford [ICFP’02, POPL’04]
  - Pappy: A packrat parser generator for Haskell
- Parses top-down (like LL)
- Treats every character as a token (unlike LR and LL)
- Performs backtracking, but memoizes each result
  - Linear time performance
  - One large table: characters on x-axis, nonterminals on y-axis

Features unique to Rats! marked by 🐭
So, Why Not LR or LL Parsing?

- LR parsers are brittle in the face of change
  - E.g., synchronization on multiple objects [Brabrand ‘03]
    - “synchronized” (‘ Expression Expression ‘)’ Block
    - 29 shift/reduce and 26 reduce/reduce errors
- LL parsers require lookahead to disambiguate
  - Factor common prefixes by hand
  - JavaCC: Use explicit lookahead expressions
  - ANTLR: Set global lookahead flag, fine-tune local options
What About the Separation Between Lexers and Parsers?

- Can be hard to add new tokens
  - Example: Regex-like character classes (think [0-9a-fA-F])
    - Want: ‘[‘ (Char ‘-’ Char / Char )* ‘]’
    - Get: Lots of ambiguity errors b/c Char overlaps (most) other tokens
  - (Tedious) workarounds
    - Use different lexer states
    - Use different lexers (ANTLR)
An Example

- LR parsers: Keep on shifting
- LL parsers: Look ahead
- Packrat parsers
  - Parse EqualityExpression, store result in table
  - Parse Symbol, store result in table
  - Compare Symbol to “&”
    - On match, continue with AndExpression
    - On mismatch, return stored result for EqualityExpression

```
void AndExpression =
  EqualityExpression "&" : Symbol AndExpression
/ EqualityExpression ;
```
Rats! Grammars

- **Header**
  - Package and class declarations
  - Code inclusions (before, inside, and after parser class)
  - Top-level nonterminals

- **Productions**
  - \( \text{Type Nonterminal} = \text{Expression} \) ;
Expressions and Operators

- Ordered choices: $e_1 / e_2$
- Sequences: $e_1 e_2$
- Prefix operators
  - Predicates: $&e$, $!e$, $&\{…\}$
  - Bindings: $id:e$
  - String matches: “…”:$e$
- Suffix operators
  - Options: $e?$
  - Repetitions: $e^*, e^+$

Primary expressions

- Nonterminals
- Terminals
  - Any character constant: .
  - Character literals: ‘a’
  - String literals: “abc”
  - Character classes: [0-9]
- Actions: { yyValue = null; }
- Grouping: ( $e$ )
Determining Semantic Values

- Through an action

  Production Production =
  type:QualifiedName nt:Nonterminal Assign choice:Choice Semicolon
  { yyValue = new Production(nt, type, choice); } ;

- By passing the value through

  Action Header =
  “header” Spacing yyValue:Action ;

- By letting *Rats!* deduce the value

  Element Primary =
  Nonterminal / Terminal / Action / OpenParen Choice CloseParen ;
Void and Text-Only Productions

- **Goal:** Avoid specifying semantic values by hand

- **Void productions**
  - Specify type “void,” have null as semantic value
    ```
    void OpenParen = '(' Spacing ;
    ```
  - Enable deduction of semantic value in other productions
    ```
    Element Primary = … / OpenParen Choice CloseParen ;
    ```

- **Text-only productions**
  - Have type “String,” contain no actions, may only reference text-only productions
    ```
    String StringLiteral = [“] (EscapeSequence / ![“\] .)* [“] ;
    ```
  - Have matched text as semantic value
Options, Repetitions, and Nested Choices

- **Issues**
  - Intermediate results need to be correctly memoized
  - Parser code generation needs to be correct & manageable
- **Options, repetitions, and nested choices are lifted**
  - Only where necessary; combined when possible
- **Options and repetitions are also desugared**
  - $nt = e? \rightarrow nt = e / ;$
    - Semantic value for second alternative is null
  - $nt = e^* \rightarrow nt = e nt / ;$
  - $nt = e^+ \rightarrow nt = e nt / e ;$
    - Semantic value is a (Scheme-like) list of individual semantic values
Error Handling

- Performs consistency checking for grammars
  - Notably, detects and flags (indirect) left-recursion
- Prints grammars after parsing and optimizations
  - Users can individually select optimizations
- Enforces declared type of productions
  - yyValue and bindings use declared type, though semantic values use type erasure [Bracha ‘98]
    - Common class SemanticValue declares value as “Object”
- Annotates AST nodes with location information
  - File name, line and column numbers
Error Handling (cont.)

- Deduces error message from nonterminals:
  - E.g., StringLiteral ➔ “string literal expected”
  - Reported position is start of production
    - For string literals and string matches, position of literal/match

- Collects parse errors even for successful parser steps
  - Higher-level parse errors may be less specific
  - Consider: Production* EndOfFile
    - Production* is always successful
    - But, if there is an error while parsing a single Production, EndOfFile will fail with a non-descriptive error message
Motivation and vision for xtc
Introduction to Rats!, our parser generator
Optimizations
Evaluation
Our compiler framework so far
Future work and conclusions
Structure of Parsers

- Parser objects represent table columns
  - Methods parse productions
    - Each sequence corresponds to several nested if statements
  - Fields memoize results
    - Null indicates that the production has not been tried before
- Result objects represent memoized results
  - Virtual methods provide uniform access to data
  - SemanticValue represents successful parses
    - <actual value, pointer to next parser object, possible parse error>
  - ParseError represents failed parses
    - <error message, pointer to parser object>
Basic Optimizations

- **Chunking**
  - Use a level of indirection to reduce space for fields
  - Do not allocate a field, if production is only used once

- **Duplicate folding**
  - Fold equivalent productions into a single one
    - Reduces overhead of lifted options, repetitions, and choices

- **Dead production elimination**
  - Remove unused (non-top-level) productions from grammar

- **Cost-based inlining**
  - Inline trivial productions
Transient Productions

- Insight: Most token-level productions never backtrack
  - Keywords and identifiers
  - Symbols
  - Spacing (and there’s lots of it!)
  - But typically not numbers

- Give grammar writers control over memoization
  - “transient” keyword disables memoization

```
transient void Spacing =
    ( WhiteSpace / TraditionalComment / EndOfLineComment )* ;
```

- Use with care: Inspect grammar or measure effects
  - In practice, build on common token-level productions
Choice Inlining and Terminal Recognition

- Insight: Many token-level productions have alternatives that start with different characters
- Inline only nonterminal in an alternative
  - Only for transient productions to preserve contract
- Use switch statements for disjoint alternatives
  - Combine common prefixes
- Avoid dynamic instantiation of matched text
  - Use string if text can be statically determined
  - Use null if the text is never used (i.e., bound)
Suppression of Unnecessary Errors and Semantic Values

- Insight: Most alternatives fail on the first expression
  - Example: Statement production for C, C++, Java, etc.
  - Only create ParseErrors for subsequent expressions and the overall production (if all alternatives fail)

- Insight: Many productions pass the value through
  - Example: 17 levels of expressions for Java, all of which must be invoked to parse a literal, identifier, etc.
  - Only create a new SemanticValue if the contents differ (otherwise, reuse the passed-through value)

- Also, combine common prefixes to avoid repeated method calls and possibly memoization
Repetitions in Transient Productions

- Observation: Desugared repetitions recurse (deeply)
  - Remember that $nt = e^* \Rightarrow nt = e \ nt /$ ;
  - This may cause stack overflows for deep recursions
    - In our experience, for very long comments
  - But transient productions are not memoized

- Preserve repetitions in transient productions
  - Complicates parser code generation
  - Ensures scalability over long inputs
  - Avoids creation of intermediate semantic values
Motivation and vision for xtc
Introduction to *Rats!*, our parser generator
Optimizations
Evaluation
Our compiler framework so far
Future work and conclusions
Evaluation

- Relevant metrics
  - Overall memory consumption
  - Overall performance
  - Effect of individual optimizations

- Experimental methodology
  - 800 MHz iMac (iLamp) vs. 3 GHz Dell Dimension 8250
  - Java parsers generated by Rats!, ANTLR, and JavaCC
    - Collect text for token-level expressions, but no semantic values
  - Java source files taken from Cryptix, Rats!, and ANTLR
  - Measurements are average of 20 iterations (+ 2 to warm up)
    - Only measure latency, memory utilization of parsing
Heap Utilization

File Size (KB) vs Heap Utilization (KB)

- Rats! 113:1
- ANTLR 11:1
- JavaCC 10:1
Performance (iMac)

File Size (KB) vs. Latency (ms) for different tools:

- **ANTLR 588 KB/sec**
- **JavaCC 1316 KB/sec**
- **Rats! 200 KB/sec**
Performance (Windows)

![Graph showing performance comparison between Rats!, ANTLR, and JavaCC for file sizes ranging from 0 to 80 KB. The graph includes markers for each tool with corresponding latency values in milliseconds (ms).]
Effect of Optimizations

Collected on iMac, using Rats’ own parser generator as input
Discussion of Results

- *Rats!* requires more resources than ANTLR, JavaCC
  - With prefix folding (not shown in graphs) up to 121:1
  - But also simpler and more flexible specification
    - *Rats!* 530 lines, ANTLR and JavaCC each 1200 lines
  - Has respectable *absolute* performance (67 KB in 0.3 sec)
- Results compare favorably with Ford’s experiments
  - Written in Haskell, using 1.3 GHz Athlon PC running Linux
  - Construct full AST, measure complete runtime
    - *Rats!*: 113:1, 200 KB/sec
    - Pappy: 441:1, 43.4 KB/sec
    - Handwritten: 297:1, 52.1 KB/sec
Talk Outline

- Motivation and vision for xtc
- Introduction to Rats!, our parser generator
- Optimizations
- Evaluation
- Our compiler framework so far
- Future work and conclusions
Our Framework So Far

- **Three main abstractions**
  - Abstract syntax tree (AST) nodes to represent code
  - Visitors to walk and transform AST nodes
    - Methods selected through reflection (with caching)
  - Utilities to store cross-visitor state
    - Analyzer for analyzing grammars
    - Printer for pretty printing source code

- **Two axes of extensibility**
  - New visitors to represent new compiler phases
  - New AST nodes to represent new language constructs
    - Corresponding visit() methods specified with new AST nodes
Future Work

- **Rats!**
  - Automatic tree generation, based on generic AST node
  - Further optimizations by not allocating parser objects

- **Domain-specific languages**
  - Syntax specification and desugaring
  - Typing constraints

- **Port to xtc**
  - Definition of object model, implementation of collections
  - Actual port
Conclusions

- System builders need better language tools
- We started building an extensible compiler toolkit
  - Basic framework
  - Parser generator
- Our parser generator improves on Ford’s work
  - Simpler grammars through void, text-only productions
  - Easier to debug, better error reporting
    - Prints grammars, deduces error messages, includes source locations
  - Better performance through aggressive optimizations