xtc

Towards eXTensible C

Robert Grimm
New York University

The Problem

- Complexity of modern systems is staggering
  - Increasingly, a seamless, global computing environment
- System builders continue to use C (or C++)
  - Inadequate basis for coping with this complexity
    - There simply is too much code (which is unsafe to boot)
  - As a result, software quality and security suffer
    - Think critical updates for Windows or Mac OS X
- *Metaprogramming* holds considerable promise
  - Basic idea: Automatically compute (parts of) programs instead of manually writing them

The Power of Metaprogramming

- libasync [Mazieres, Usenix ’01]
  - Provides extensions for asynchronous programming
    - Helps write straight-line code in presence of callbacks
- Capriccio [von Behren et al., SOSP ’03]
  - Provides a highly scalable threading package
    - Uses stack depth analysis to reduce memory overhead
- MACEDON [Rodriguez et al., NSDI ’04]
  - Provides DSL for specifying overlay protocols
    - Compiles state machine specification to runnable system

➜ How to give this power to other system builders?
  - Make language and compiler extensible through macros!

The Four Requirements

- *Expressive* enough for new definitional constructs
  - General (e.g., modules, objects, generics,…)
  - Domain-specific (e.g., closures, state machines)

```
char* host = “www.example.com”;
char* path = “/index.html”;
dnslookup(host, void closure(ip_t addr) {
  tcpconnect(addr, 80, void closure(int fd) {
    … write(fd, path, strlen(path)); …
  });
});
```

- *Safe* to statically detect program errors
  - Not only macro hygiene, but also new typing constraints
    - Avoid string of obscure error messages (e.g., C++ templates)

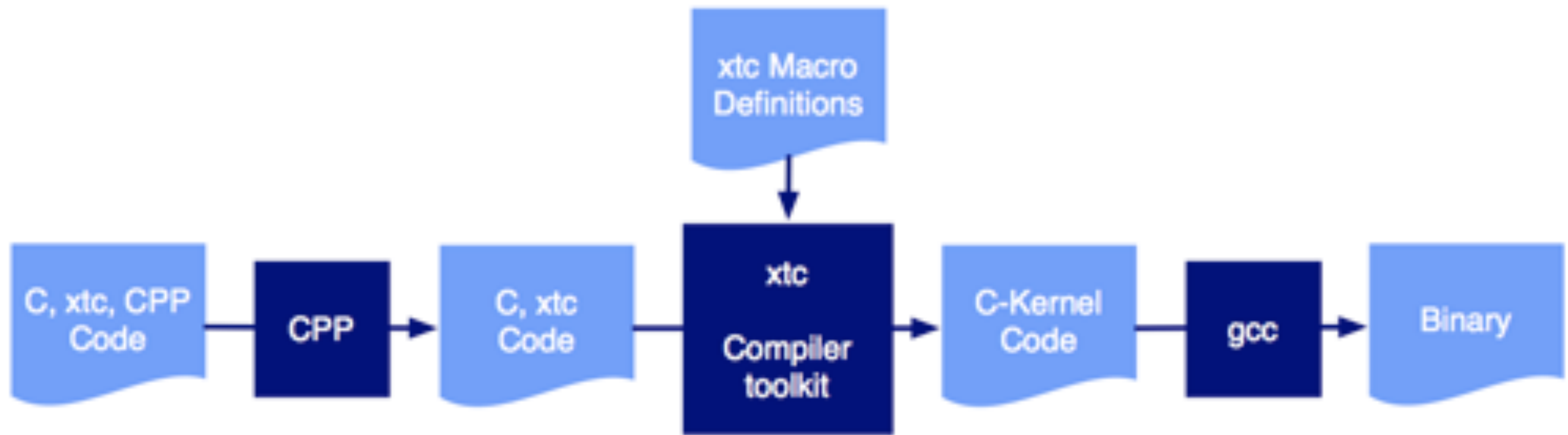
The Four Requirements (cont.)

- *Fine-grained* to compose and reuse extensions
  - Track dependencies
  - Detect conflicts
- *Efficient* code as compiler output
  - Specialized expansions (e.g., `foreach` on arrays)
  - Domain-specific optimizations
    - E.g., stack depth analysis for Capriccio

➜No existing macro system meets all requirements

- Little work on extensible type systems, macro composition
- Little work on C (or C++) with all its particularities

Enter xtc (eXTensible C)



- Opens both language and compiler through macros
- Includes (meta-) module system
- Leaves conventional optimizations and code generation to regular compiler (gcc)
- Implemented on top of toolkit for extensible source-to-source transformers

More Details on xtc Macros

- Expressed as declarative rules
  - Grammar modifications
  - Abstract syntax tree (AST) transformations
    - Based on templates that mix literal syntax with pattern expressions
  - Type constraints
    - To add new types, control subtyping, specify type of expressions
- Selected through multiple dispatch [Baker, PLDI '02]
  - Including the types and kinds of types of pattern var's
    - Types for optimized expansions, kinds for new definitional constructs
- Target CIL-like C-kernel as base language
  - Simplifies optimizations by removing redundancy

Talk Outline

- Motivation and vision for `xtc`
- Introduction to our toolkit, including parser generator
- Functional parsing with state
- Parser optimizations
- Results
- Extensible syntax
- Our AST framework and query engine
- Conclusions and outlook

Our Prototype Toolkit

- Focused on extensibility and ease-of-use
  - Easy to change grammars, source-to-source transformers
    - Changes should be localized (e.g., avoid grammar-wide conflicts)
    - Toolkit buy-in should be minimized (i.e., use only what you need)
  - In contrast, existing compilers are complex, hard to extend
    - LALR grammars are brittle under change
    - Generalized LR parsers require disambiguation
  - Need to rebuild language tool chain from the ground up
- Includes parser generator, abstract syntax tree (AST) framework and query engine

Our Prototype Toolkit (cont.)

- Written in Java
  - Represents *temporary* engineering trade-off
    - Simple object system, GC, collections framework, reflection
  - Provides us with a first, large-scale test case
    - As soon as `xtc` is sufficiently functional, will make it self-hosting
- Released as open source (currently, version 1.6.1)
  - Includes real programming language support
    - Parser and pretty printer for C
      - Processes entire Linux kernel
    - Parser for Java (with pretty printer coming "soon")
    - Parser and desugarer for aspect-enhanced C in the making

Rats! Parser Generator

- Basic strategy: Packrat parsing
  - Originally described by Birman [Phd ‘70]
  - Revisited by Ford [ICFP’02, POPL’04]
    - Pappy: A packrat parser generator for Haskell [Masters ‘02]
  - Parses top-down (like LL)
  - Orders choices (unlike LR and LL)
  - Treats every character as a token (unlike LR and LL)
  - Supports unlimited lookahead through syntactic predicates
  - Performs backtracking, but memoizes each result
    - Linear time performance
    - One large table: characters on x-axis, nonterminals on y-axis

Rats! Grammars

- Header
  - Grammar wide attributes (including parser class name)
  - Code inclusions (before, inside, and after parser class)
- Productions
  - *Attribute* Type Nonterminal* = *Expression* ;

Expressions and Operators

- Ordered choices: e_1 / e_2
- Sequences: e_1 e_2
- Voiders: void:*e*
- Prefix operators
  - Predicates: &*e*, !*e*, &{…}
  - Bindings: *id*:*e*
  - String matches: "…":*e*
  - Parser actions: ^{…}
- Suffix operators
  - Options: *e*?
  - Repetitions: *e**, *e*+
- Primary expressions
  - Nonterminals
  - Terminals
    - Any character constant: _
    - Character literals: 'a'
    - String literals: "abc"
    - Character classes: [0-9]
  - Semantic actions
    - { yyValue = null; }
  - Grouping: (*e*)

Ordered choices, predicates, EBNF operators, semantic values

System Programmers Are Lazy Or, How to Avoid Semantic Actions

- Void productions
  - Specify type "void," have null as semantic value

```
void Space = ' ' / '\t' / '\f' ;
```

- Text-only productions
  - Specify type "String," have matched text as value

```
String StringLiteral = ["] (EscapeCharacter / !["\\] _)* ["] ;
```

- Generic productions
  - Specify type "generic," have generic AST node as value
    - Children are semantic values of component expressions

```
generic Declaration = DeclarationSpecifiers
                      InitializedDeclaratorList? void:";":Symbol ;
```

Talk Outline

- Motivation and vision for `xtc`
- Introduction to our toolkit, including parser generator
- Functional parsing with state
- Parser optimizations
- Results
- Extensible syntax
- Our AST framework and query engine
- Conclusions and outlook

Functional Parsing with State

- Problem: Packrat parsers are basically functional
  - But we need a symbol table for parsing C
    - Typically, shared between lexer and parser ("lexer hack")
- Naïve solution: Throw away memoized results
  - Expensive, violates linear time guarantees
- Potential, established solution: Monads
  - Have memory and performance overheads [Ford '02]
  - Are not supported by C-like languages
  - Are not familiar to our target audience (system builders)

➜Can we do better?

Functional Parsing with State: Transactions to the Rescue

- Insight (valid for computer languages/file formats)
  - State has well-defined (nested) scope, flows forward
- Practical solution: Lightweight transactions
  - Bracket alternatives that might modify state in transactions
  - Modify state in common prefixes in exactly the same way

```
GNode ExternalDeclaration = { yyState.start(); }
    Declaration                   { yyState.commit(); }
  / FunctionDefinition            { yyState.commit(); }
  / { yyState.abort(); } &{ false } ;
```

  - Transactions easily implemented by pushing/popping contexts
    - For C, each context (potentially) represents a symbol table scope
  - *Rats!* automatically generates calls based on “stateful” attribute

Functional Parsing with State: Transactions to the Rescue

- Insight (valid for computer languages/file formats)
  - State has well-defined (nested) scope, flows forward
- Practical solution: Lightweight transactions
  - Bracket alternatives that might modify state in transactions
  - Modify state in common prefixes in exactly the same way

```
stateful GNode ExternalDeclaration =
    Declaration
  / FunctionDefinition
  ;
```

  - Transactions easily implemented by pushing/popping contexts
    - For C, each context (potentially) represents a symbol table scope
  - *Rats!* automatically generates calls based on “stateful” attribute

The Symbol Table Is Not Enough Or, The Devil Is in C

- Beware of functions redefining type names

```
typedef int number;
number function(number number) { … }
```

  - Solution: Track type specifiers
- Beware of functions returning functions

```
typedef int number;
int (* function(int argument))(int number) { … }
```

  - Solution: Track function declarators
- Solutions easily implemented as per-context flags

It Gets Worse: A Tale of Three Cs (ISO, K&R, GCC)

- Some GCC attributes can cause very far lookahead

```
void _exit(int) __attribute__((__noreturn__));
```

  - Attribute parsed as declaration (in K&R function definition)

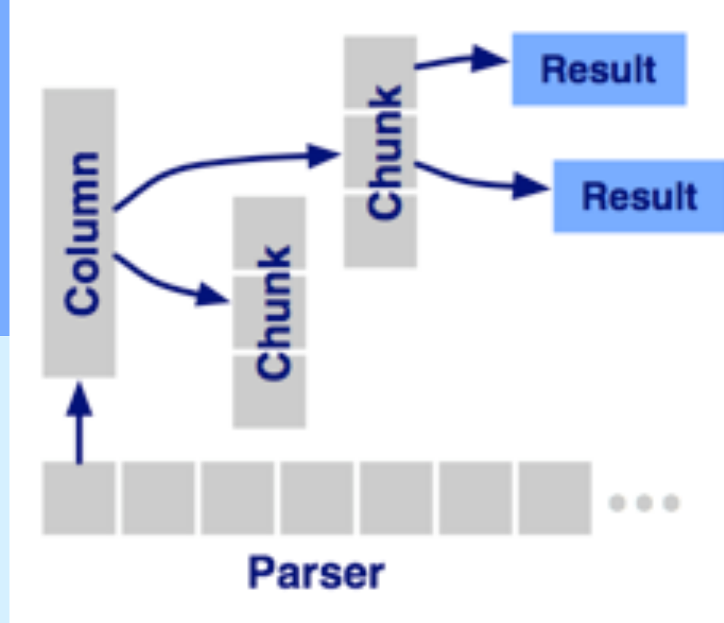
```
DeclarationSpecifiers InitializedDeclaratorList? ";":Symbol
```

  - Functional parsing works
    - Parser simply backtracks when not finding a compound statement
  - State still gets corrupted
    - GCC line markers are parsed as spacing and recorded in state
    - But far lookahead gobbles them up
- Solution: Leverage existing state
  - Disallow empty declarator list without a type specifier

Talk Outline

- Motivation and vision for `xtc`
- Introduction to our toolkit, including parser generator
- Functional parsing with state
- Parser optimizations
- Results
- Extensible syntax
- Our AST framework and query engine
- Conclusions and outlook

Implementation of Parsers



- Parser obj's methods represent productions
  - Each sequence implemented by nested if statements
- Array of chunked column obj's memoizes 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, index of next column, possible parse error>
  - ParseError represents failed parses
    - <error message, index of error>

Transient Productions: Avoiding Memoization

- Insight: Most productions are never backtracked over
  - Token-level: Keywords, identifiers, symbols, spacing
    - But typically *not* numbers
  - Hierarchical syntax: look at tokens before each reference
    - If different, production is not backtracked over for input position
- Give grammar writers control over memoization
  - “transient” attribute disables memoization
    - Doubly effective: Eliminates rows and columns from memo table
- Enable further optimizations
  - Preserve repetitions as iterations
  - Turn direct left-recursions into equivalent right iterations

Improved Terminal Recognition: Inline, Combine, and Switch

- Insight: Many token-level productions have alternatives that start with different characters
  - Inline only nonterminal in an alternative
    - Only inline transient productions to preserve contract
  - 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 the text is never used (i.e., bound)

Suppression of 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 a new SemanticValue if the contents differ
    (otherwise, reuse the passed-through value)
- Insight: Most alternatives fail on first expression
  - Example: Statement productions for C, C++, Java, etc.
  - Only create a new ParseError if subsequent expressions
    or all alternatives fail
    - Meanwhile, use a generic error object

Talk Outline

- Motivation and vision for `xtc`
- Introduction to our toolkit, including parser generator
- Functional parsing with state
- Parser optimizations
- Results
- Extensible syntax
- Our AST framework and query engine
- Conclusions and outlook

Rats! Has Reasonable Performance

- Evaluated performance of Java parsers
  - Only lexing and parsing, no AST construction
- *Rats!* requires more resources than ANTLR, JavaCC
  - 1.4–3 times slower, 4 times as much memory
  - But absolute performance is pretty good: 67 KB in 0.18s
- Results compare favorably with Ford’s work
  - All of Pappy’s optimizations supported by *Rats!*
  - 83.6% speedup from optimizations, 46.7% is new
  - In absolute terms, 8.8 times faster on similar hardware
- Future work
  - Further optimizations, port to `xtc` (no more JITed VM)

Rats! Has Concise Grammars

	Lexer	Parser
xtc		1,160
C: CIL	600	1,400
gcc	800	3,500

	Parser
Rats!	530
Java: ANTLR	1200
JavaCC	1200

- Also, good support for debugging and error reporting
  - Pretty-printing of grammars and parsers
  - Automatic annotation of AST with source locations
  - Automatic generation of error messages
    - E.g., StringLiteral ➜ “string literal expected”

Talk Outline

- Motivation and vision for `xtc`
- Introduction to our toolkit, including parser generator
- Functional parsing with state
- Parser optimizations
- Results
- Extensible syntax
- Our AST framework and query engine
- Conclusions and outlook

Extensible Syntax

- *Rats!* performs good enough, has concise grammars, and supports *localized* changes
  - Foundation for truly extensible syntax
- Modules factor units of syntax
  - Group several productions
  - May depend on other modules (the "imports")
- Module modifications specify language extensions
  - Map one module to another (think functor for syntax)
  - Add, override, remove alternatives from productions
  - Add new productions

Extensible Syntax (cont.)

- Module parameters support composition
  - Specify module names (instantiated from top down)
    - Can use the same module with different imports, modified modules
  - Provide “scalable extensibility” [Nystrom, CC ’03]
- This has been tried before, with major short-comings
  - Cardelli et al. on “extensible syntax”
    - LL(1), no parameterized modules
  - Visser et al. on SDF2 and Stratego/XT
    - Generalized LR (with lots of disambiguation), no production changes, module parameters only for types of semantic values

Talk Outline

- Motivation and vision for `xtc`
- Introduction to our toolkit, including parser generator
- Functional parsing with state
- Parser optimizations
- Results
- Extensible syntax
- Our AST framework and query engine
- Conclusions and outlook

Our Framework So Far

- Three main abstractions
  - Abstract syntax tree (AST) nodes to represent code
  - Visitors to walk and transform AST nodes
    - Methods dynamically 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
    - `process()` methods specified with new AST nodes

Is This Enough?

- Our AST framework is simple and extensible
  - Roughly, comparable to Polyglot [Nystrom, CC '03] and CIL [Necula, CC '02]
  - But there is so much more to do…
- Basic desugaring: C to C-kernel
- Macros introducing new definitional constructs
  - E.g., object macro treats fields differently from methods
- Multiple dispatch macro selection
  - Including access to type and kind attributes

Enter the AST Query Engine

- Language modeled after XPath
  - Combines simple, path-like selectors with flexible filter expressions
  - Adds support for bindings, templates, replacements
- Supported by library for analyzing ASTs
  - E.g., determine free variables, track type information
- Queries may be scheduled concurrently
  - Trigger only first matching query
    - Multiple dispatch for macros

Talk Outline

- Motivation and vision for `xtc`
- Introduction to our toolkit, including parser generator
- Functional parsing with state
- Parser optimizations
- Results
- Extensible syntax
- Our AST framework and query engine
- Conclusions and outlook

Conclusions

- Metaprogramming can help us cope with increasing complexity of (distributed) systems
  - Need expressiveness, safety, composability, efficiency
- `xtc` strives to meet these requirements for C
  - Expresses macros as collections of rules
  - Builds on our toolkit for source-to-source transformers
    - *Rats!*, our practical packrat parser generator
    - Module system (initially) for syntax
    - AST nodes and (dynamically dispatched) visitors
    - AST query engine

Outlook

- Current focus
  - Completion of module system for *Rats!*
  - Improvements and evaluation of query engine
  - Improvements to Java support
  - End of summer: start porting `xtc` over to itself
    - Useful even without type safety and macro module system
- Also, exploring an extensible type system
  - We can build on CCured [Necula, POPL '02] and similar work
  - How to ensure safety of transformations?
  - How to ensure soundness of type system extensions?

Outlook (cont.)

- Open question: How to transform from C to C?
  - Without preprocessing first
  - Example: Aspect-enhanced version of Linux
    - Manage kernel variations at semantic level
      - Not textual line-by-line level (diff/patch)

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