Grammar
Programming in TXL

Thomas Dean
James Cordy
Queen’s University
Kevin Schneider
University of Saskatchewan
Andrew Malton
University of Waterloo
Initial Comments

- A lot of source code analysis and manipulation tools parse and transform the input
  - Craft a grammar
  - Parse input to tree
  - Rule sets use the grammar to structure the transform
Parsing Technologies

- We’ve come a long way since lex and yacc
- Generalized Parsing Implementations
  - TXL [Cordy91, Cordy00]
  - GLR (ASF+SDF) [Rekers92, van den Brand et al. 98]
- Grammars not restricted to LALR(1) or LL(1)
Consequences

- Grammars closer to “natural” structure of language
  - Easier to write grammars
  - Easier to write transforms (correctly)
- Easier to modify
  - Language dialects (microsoft C vs gcc)
  - Sublanguages (embedded SQL)
If we can modify the grammar to add language dialects and sublanguages, why not change the grammar for an individual tool?

Grammar Programming

Let the Parser Do The Work!!
Grammar Programming

- Experienced TXL programmers are just as likely to change the grammar as they are to write a rule

- Paper identifies 5 general techniques
Core Concepts

- **Base grammar**
  - Good for general transformations
  - Easily modified for custom transformations

- **Transform Specific Overrides**
  - Recatagorize syntactic elements (C typedef)
  - Add loopholes in the grammar for transforms
Transform Architecture

- One Grammar
  - All transforms have to conform to the global grammar
Multiple transform sets

- Each with their own custom grammar
- Back to source code between each step
Example - Unique Naming

- Give each entity a unique name
  - Subject of future paper
- Markup in the code
- Base grammar requires markup
  - Original code does not
- Unique name transform has modified grammar that makes unique name markup optional
Parsing Speed

- Parser generation is fast
- Parser operation is fast
- Every time we used one of the techniques outlined in the paper, our overall time went down
- Parsing is Free!!