Parsing with IRONY

http://www.codeplex.com/irony

Roman Ivantsov, MCSD
Software Architect, Tyler Technologies, Eden Division
www.tylertech.com
roman.ivantsov@tylertech.com

Agenda

- Introduction and background
- Demo - Expression Grammar
- Inside Irony – internal architecture overview
- Future plans

Introduction

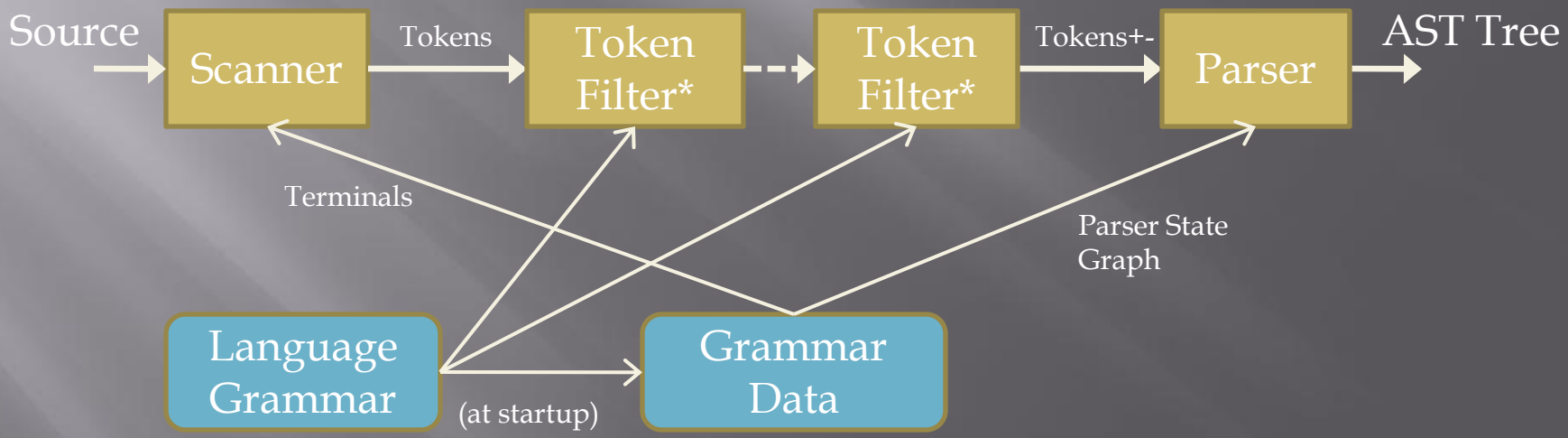
- Name
- Motivation - upgrade compiler construction technology.
- Goal – make it easier and make it faster
- Feature set – whatever Lex/Yacc can do and more
- Method
  - All in c#: embedded Domain-Specific Language (DSL) for grammar specifications
  - No code generation: one-for-all LALR(1) Parser
  - Scanner construction from standard and custom token recognizers

Demo

- Grammar for Arithmetic Expressions
- Grammar Explorer
- Sample Grammars

Irony Parsing Architecture

Parsing Pipeline



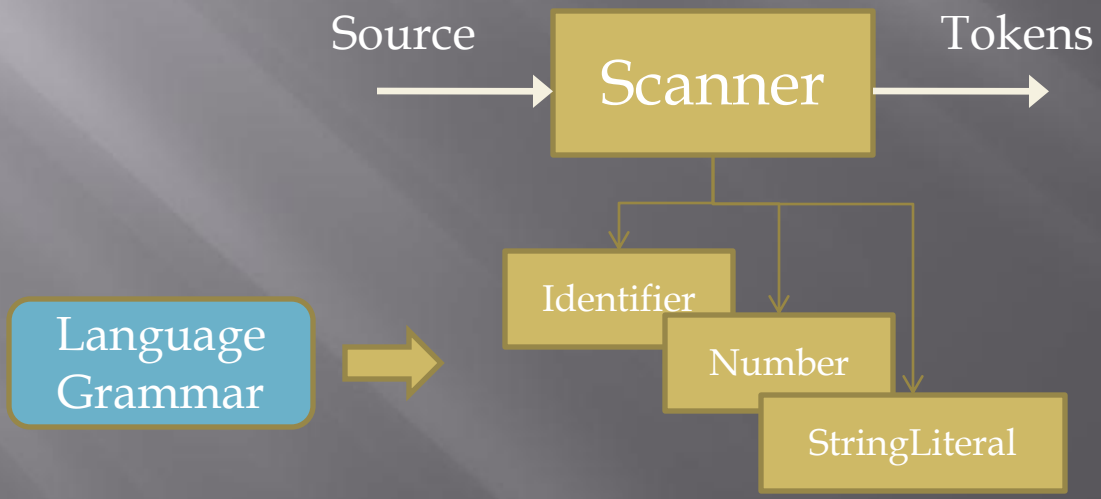
** - optional; standard or custom filter*

Grammar

- Base class for language grammars
- c# operator overloads defined in BnfElement class; Terminal and NonTerminal are derived from BnfElement
- Helper methods for Kleene operators * + ?; list constructor with optional delimiters

Scanner

- OOP-style micro-framework for building custom scanners from standard and custom Terminals
- Terminal is a token recognizer
- Highly-optimized for performance
- Ignores whitespace; new-line, indent and unindent tokens are created in token filter if necessary



Token Filters

- Sit between Scanner and Parser
- Perform language-specific tasks:
  - Code outlining - create NewLine, Indent, Unindent, EndOfStatement tokens
  - Commented-out blocks
  - Macro-expansion
  - Checking matching pairs of braces/parenthesis
  - Handling XML documentation

Parser

- LALR(1) algorithm controlled by State Graph built at startup
- AST node type determined by NodeType property of NonTerminal in language grammar.
- Customization
  - Custom AST nodes
  - Factory methods for node creation
  - Public events and overridable methods in grammar

Highlights

- It works!
- Easier to use, expect shorter implementation time
- High performance: 10..20 K lines/second
- High code reuse – reusable terminals, token filters, AST nodes
- No generated code => higher code quality + easier to maintain

Future Development

- Completing features for 1.0 release (est. Feb 2008)
- Create a set of standard AST nodes
- Extending to LALR(1.5) parser – with extra token preview
- Create basic infrastructure for building interpreters
- Build Runtime
  - Extensible Object Model (Piumarta, Warth)
  - Type model (yes, wrappers!)
- Simple scripting language(s) for testing