Modeling Software Structures with GrammarLab

MoDELS 2013 Tutorial
Vadim Zaytsev, SWAT, CWI
© 2013
Dr. Vadim Zaytsev
Dr. Vadim Zaytsev
Technical side of bridging

Javier Cánovas, Integrating APIs in the modeling realm,


• ...


• ...
To summarise

- Real world is shared
- Models Grammars everywhere
- Reuse across TSs
- Methodology sharing

http://commons.wikimedia.org/wiki/File:Torii_kiyoshige_bando_hikosaburo_ii.jpg
Tutorial ::= intro grammars inside outside rascal grammarlab
Grammars in a broad sense

Grammars in history

- Panini (IV BC)
  - Ashtadhyayi grammar
- Chomsky (1956)
  - hierarchy of grammars
- Backus (1959), Naur (1963), Wirth (1977)
  - notation for grammars
URGH! THRAK CREATE FIRE! FIRE HOT!

FIRE NEAT, THRAK, BUT ME INVENT CLUB! CLUB FOR HIT THINGS WITH.

HELLO, FRIENDS! IT SEEMS THAT I HAVE INVENTED GRAMMAR.

THANKS! ME AND THRAK CAN COMMUNICATE FAR MORE EASILY NOW THAT WE HAVE HELPING VERBS AND ARTICLE ADJECTIVES.

"THRAK AND I"
Grammars in SE

- Definitions of languages
  - Finite definitions
  - of infinite languages
- Focus on sets of strings ("words")
- Also, rewriting systems
- Analytic or generative
Arithmetic Expressions, Boolean Expressions, and Expressions

\[\text{exp} \equiv \text{ar exp} \lor \text{bool exp}\]
1.5 Greibach normal form

Problem 5.

Is it true that for every Boolean grammar there exists a Boolean grammar in Greibach normal form that generates the same language?

For a small special case the question has been answered affirmatively:

Theorem 5

Alexander Okhotin, Nine open problems on conjunctive and Boolean grammars

Theorem 5

nOkhotinr Reitwießner [7].

For every conjunctive grammar over a unary alphabet there exists and can be effectively constructed a conjunctive grammar in Greibach normal form generating the same language.

One previously given advice must be revoked. It was suggested to try the following language:

The question whether the language \{a^n b^{2n} | n \geq v}\ can be represented by a Boolean grammar in Greibach normal form might be a good starting point in approaching Problem 5. The answer to this question is likely negative, and a negative solution to the problem can be thus obtained.

Surprisingly, there exists a conjunctive grammar for this language:

Example 3.

The following conjunctive grammar generates the language \{a^n b^{2n} | n \geq v}\:

\[
\begin{align*}
S & \rightarrow aSB \& AB \mid b \\
A & \rightarrow aA \mid \varepsilon \\
B & \rightarrow B_1 \mid B_2 \\
B_1 & \rightarrow B_1 B_3 \& B_2 B_2 \mid b \\
B_2 & \rightarrow B_1 B_1 \& B_2 B_6 \mid bb \\
B_3 & \rightarrow B_1 B_2 \& B_6 B_6 \mid bbb \\
B_6 & \rightarrow B_1 B_2 \& B_3 B_3
\end{align*}
\]

Each nonterminal $B_i$ generates \{b^{4n} | n \geq v\} as in Example 3 and in particular $B_2$ generates \{b^{2n} | n \geq v\}.

All strings of the form $a^n b^{2n}$ are generated inductively by $S$.

The basis is the string $b = a^0 b^{20}$ generated by the rule $S \rightarrow b$.

The rule $S \rightarrow aSB \& AB$ generates all strings of the form $a^{n+1} b^{2n+i}$ in which both $x^n q_i$ and $i$ are powers of two.

These conditions are satisfied only if $i = x^n q_i$ and thus the generated strings must be of the form $a^{n+1} b^{2n+1}$.

1.6 Complementation of conjunctive grammars

Problem 6.

Is the family of conjunctive languages closed under complementation?

It is worth being added that the same problem could be separately considered for conjunctive languages over a unary alphabet. In fact, for every unary language found to be conjunctive in the literature its complement could be proved conjunctive by the same methods [xr y]. Yet there are no methods of changing a given conjunctive grammar to a grammar for the complement of the generated language.
Grammars as contracts

- Many metatools
  - each implies its own language
- Interoperability
  - depends on conforming to the same language

4.1

manipulate, representation

This is exact definition.

The possible actions are:

1. `parsetree2ast` will generate `AST`
2. `ast2parsetree` will generate `AST`

The relationship between concrete syntax and abstract syntax is as follows:

- Concrete syntax definition is manipulated into abstract syntax definition.
- `parsetree2ast` converts parse trees to abstract syntax trees.
- `ast2parsetree` converts abstract syntax trees to parse trees.

These tools are used for creating libraries and generators.
Grammars as commitments

- Intercommunication purposes
- Bridging-in-the-small
  - Assume two independent collaborators
  - Each side has its own infrastructure
  - E.g., XML and ANTLR
  - Information exchange is crucial
  - The system is evolving
Perceived view "Lines of code" | Proposed view "Impact ratio"

All software assets

Grammars in compiler front-ends

Grammars in a broad sense

Ingrained grammar dependencies

5. PROMISES OF GRAMMARWARE ENGINEERING

At this point, the reader might face the following question: Somehow we managed to deal with all these kinds of grammarware. So what? That is, what are the potential benefits for IT?

The overall promise of grammarware engineering is that it leads to improved quality of grammarware and to increased productivity of grammarware development. These promises should provide a good incentive since grammars permeate software systems and software development. Of course, it is difficult to justify such general claims at this time. To provide some concrete data, we will report on two showcases (or even success stories). Afterwards, we will identify more detailed promises on the basis of these showcases, but we will also refer to further scattered experiences with engineering aspects of grammarware.

5.1 Showcase: grammar recovery

This showcase is discussed in detail in [Lämmel and Verhoef 2001b; Lämmel 2005]. Using elements of the emerging engineering discipline for grammarware, we were able to rapidly recover a relatively correct and complete syntax definition of VS Cobol II. The starting point for this recovery project was IBM’s industrial standard for VS Cobol II [IBM Corporation 1993]. The syntax diagrams had to be extracted from the semi-formal document, and about 400 transformations were applied to the raw syntax in order to add missing constructs, to fix errors, and to ultimately obtain a grammar that could be used for parsing.
Grammars as interfaces

- Algebraic data types
- Domain models
- Class diagrams
- Libraries
- API
- Questionnaires
JavaScript
Browser runs JavaScript

CoffeeScript compiles to dart2js

is a library for jQuery

generates
EBNF
Metasyntax
Grammars
Structure

To summarise

- Grammars define languages
- Represent structural commitments
- Contracts / interfaces
- Bridgebuilding material
- Grammars in a broad sense model software languages
What is a software language?
SL is GPL

- Programming language
- General purpose
- High level constructs
- Syntactic sugar
- Can be compiled/interpreted/computed/evaluated/…
GET statuses/mentions_timeline

Updated on Thu, 2013-06-20 14:39

Returns the 20 most recent mentions (tweets containing a user’s @screen_name) for the authenticating user.

The timeline returned is the equivalent of the one seen when you view your mentions (@) on twitter.com.

This method can only return up to 800 tweets.

See Working with Timelines for instructions on traversing timelines.

Resource URL
https://api.twitter.com/1.1/statuses/mentions_timeline.json

Parameters

**count**
optional

Specifies the number of tweets to try and retrieve, up to a maximum of 200. The value of `count` is best thought of as a limit to the number of tweets to return because suspended or deleted content is removed after the count has been applied. We include retweets in the count, even if `include_rts` is not supplied. It is recommended you always send `include_rts=1` when using this API method.

**since_id**
optional

Returns results with an ID greater than (that is, more recent than) the specified ID. There are limits to the number of Tweets which can be accessed through the API. If the limit of Tweets has occurred since the since_id, the since_id will be forced to the oldest ID available.

OAuth tool

Please Sign in with your Twitter account in order to use the OAuth tool.
Data Clone Detection and Visualization in Spreadsheets

To summarise

- Language is a set of instances
- Language is modelled by a grammar
- Languages are everywhere
- Not necessarily used by experts
Abstract
&
concrete
Elements of a Modeling Language

Bran Selic, Theory and Practice of Modeling Language Design

Ralf Lämmel, http://instagram.com/p/e5P9r4TGHd/
Both are grammars!

- Concrete syntax
  - elements for visualisation & programming
- Abstract syntax
  - conceptual structure
- Beyond dychotomy!
public void Configure(Reader target) {
    target.AddStrategy(ConfigureServiceCall());
    target.AddStrategy(ConfigureUsage());
}

private ReaderStrategy ConfigureServiceCall() {
    ReaderStrategy result = new ReaderStrategy("SVCL", typeof (ServiceCall));
    result.AddFieldExtractor(4, 18, "CustomerName");
    result.AddFieldExtractor(19, 23, "CustomerID");
    result.AddFieldExtractor(24, 27, "CallTypeCode");
    result.AddFieldExtractor(28, 35, "DateOfCallString");
    return result;
}

private ReaderStrategy ConfigureUsage() {
    ReaderStrategy result = new ReaderStrategy("USGE", typeof (Usage));
    result.AddFieldExtractor(4, 8, "CustomerID");
    result.AddFieldExtractor(9, 22, "CustomerName");
    result.AddFieldExtractor(30, 30, "Cycle");
    result.AddFieldExtractor(31, 36, "ReadDate");
    return result;
}
<ReaderConfiguration>
  <Mapping Code = "SVCL" TargetClass = "dsl.ServiceCall">
    <Field name = "CustomerName" start = "4" end = "18"/>
    <Field name = "CustomerID" start = "19" end = "23"/>
    <Field name = "CallTypeCode" start = "24" end = "27"/>
    <Field name = "DateOfCallString" start = "28" end = "35"/>
  </Mapping>
  <Mapping Code = "USGE" TargetClass = "dsl.Usage">
    <Field name = "CustomerID" start = "4" end = "8"/>
    <Field name = "CustomerName" start = "9" end = "22"/>
    <Field name = "Cycle" start = "30" end = "30"/>
    <Field name = "ReadDate" start = "31" end = "36"/>
  </Mapping>
</ReaderConfiguration>
mapping SVCL dsl.ServiceCall
  4-18: CustomerName
  19-23: CustomerID
  24-27: CallTypeCode
  28-35: DateOfCallString

mapping USGE dsl.Usage
  4-8: CustomerID
  9-22: CustomerName
  30-30: Cycle
  31-36: ReadDate
Grammars on the inside
Backus Naur Form

- Historically useful
- Extended over and over
- “Backus-Wirth Notation”
- (not a normal form)
Backus Naur Form

- Historically useful
- Extended over and over
- “Backus-Wirth Notation”
- (not a normal form)
- Textual
Inside a grammar

“A ::= B “C” D? E* F+ (G I H);”

“left hand side”
A ::= B "C" D? E* F+ (G I H);
Inside a grammar

A ::= B "C" D? E* F+ (G I H);
Inside a grammar

A ::= B "C" D? E* F+ (G I H);
A ::= B "C" D? E* F+ (G I H);
A ::= B “C” D? E* F+ (G I H);
Inside a grammar

\[ X ::= \langle a \rangle : b \ [c] : : d \ ; \]
Inside a grammar

\[ X ::= \varepsilon \ ; \]

\[ X ::= \varnothing \ ; \]

\[ X ::= \alpha \ ; \]
Inside a grammar

\[ X ::= \varepsilon ; \]
\[ X ::= \varphi ; \]
\[ X ::= \alpha ; \]
Inside a grammar

\[ X ::= \varepsilon ; \]

\[ X ::= \varphi ; \]

\[ X ::= \alpha ; \]

anything
Inside a grammar

\[
X ::= \{ Y Z \}^+ ;
\]

\[
X ::= A \& B ;
\]

\[
X ::= \neg N ;
\]
Inside a grammar

\[
\begin{align*}
X & ::= \{ Y Z \}^+ . \\
X & ::= A \ & B ; \\
X & ::= \neg N ;
\end{align*}
\]
Inside a grammar

\[ X ::= \{ Y Z \}^+ ; \]

\[ X ::= A \& B ; \]

\[ X ::= \neg N ; \]
Ada 95 (Magnus Kempe, HTML)

```
compilation:
    compilation_unit

compilation_unit:
    context_clause library_item
    context_clause subunit

library_item:
    "private"? library_unit_declaration
    library_unit_body
    "private"? library_unit_renaming_declaration

library_unit_declaration:
    subprogram_declaration
    package_declaration
    generic_declaration
    generic_instantiation

library_unit_renaming_declaration:
    package_renaming_declaration
    generic_renaming_declaration
    subprogram_renaming_declaration
```

http://slps.github.io/zoo/ada/kempe.html
BibTeX (Guillaume Hillairet, Ecore)

Bibtex:

entries::Entry+

Entry:

Article

Entry:

Book

Article:

key::String fields::Field+

Book:

key::String fields::Field+

Field:

Authors

Field:

AuthorUrls

Field:

Title

http://slps.github.io/zoo/bibtex/bibtex-1.html
### ANSI C90 (Vinju, Kooiker, van den Brand, SDF)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TranslationUnit</td>
<td><code>ExternalDeclaration+</code></td>
</tr>
<tr>
<td>ExternalDeclaration</td>
<td><code>FunctionDefinition</code></td>
</tr>
<tr>
<td>ExternalDeclaration</td>
<td><code>Declaration</code></td>
</tr>
<tr>
<td>FunctionDefinition</td>
<td><code>Specifier* Declarator Declaration* &quot;{&quot; Declaration* Statement* &quot;}&quot;</code></td>
</tr>
<tr>
<td>Declaration</td>
<td><code>Specifier+ (InitDeclarator (&quot;,&quot; InitDeclarator)*) ;</code></td>
</tr>
<tr>
<td>Declaration</td>
<td><code>Specifier+ ;</code></td>
</tr>
<tr>
<td>InitDeclarator</td>
<td><code>Declarator</code></td>
</tr>
<tr>
<td>InitDeclarator</td>
<td><code>Declarator &quot;=&quot;Initializer</code></td>
</tr>
</tbody>
</table>

GNU C grammar (TXL, Andrew J. Malton, James R. Cordy)

**preprocessor:**

```
"#define" id "(" id+ ")" expression NL
"#define" id expression NL
EX ",#else" (IN NL)
EX ",#endif" (NL NL)
NL ",#if" expression (IN NL)
NL ",#ifndef" id (IN NL)
NL ",#ifdef" id (IN NL)
"#ident" stringlit NL
"#include" stringlit NL
"#include" "SPOFF file path >" SPON NL
"#line" integernumber stringlit? NL
"#undef" id NL
"#LINK" stringlit NL
```

**filepath:**

```
file_id slash_fileid*
```

**file_id:**

```
id
key
```
OASIS DocBook (RELAX-NG)

bookcomponent.content:
  divcomponent.mix+ (sect1* | refentry.class* | simplesect* | section.class*)
  sect1+
  refentry.class+
  simplesect+
  section.class+

local.set.attrib:
  
set.role.attrib:
  role.attrib

set:

set.attlist:
  fpi::string? status.attrib common.attrib set.role.attrib local.set.attrib

local.setinfo.attrib:
  
To summarise

✧ Nonterminals
✧ Terminals
✧ Metaproperties (?, *, +)
✧ Special cases
✧ Labels and markers
✧ Disjunction
✧ ...

http://commons.wikimedia.org/wiki/File:Torii_kiyoshige_bando_hikosaburo_ii.jpg
Grammars
from the outside
Grammar world

Grammar
Grammar world: grammars

Replicas
Matches
Views
Updates

Grammar
Grammar’
Grammar world: instances
Grammar world: metamodels

- Metasyntax
- Notation
- Format
- Instance
- Limitations
- Grammar’
Grammar world: annotations

- Annotation
- Metasyntax
- Documentation
- Semantic actions
- Decoration
- Grammar’
- Instance
Grammar world: transformations

- Browsable
- Annotation
- Transformers
- Metasyntax
- Extractors
- Grammar
- Exporters
- Instance
- Grammar'
- Executable
Grammar world: synchronisations

- Annotation
- Grammar
- Metasyntax
- Instance
- Grammar'
- Instance'
To summarise

- Diversity
- Evolution
- Extraction
- Documentation

http://commons.wikimedia.org/wiki/File:Torii_kiyoshige_bando_hikosaburo_ii.jpg
Grammarware
Grammarware

- Parser
- Compiler
- Interpreter
- Pretty-printer
- Scanner
- Browser
- Static checker
- Structural editor
- IDE
- DSL framework
- Preprocessor
- Postprocessor
- Model checker
- Refactorer
- Code slicer
- API
- XMLware
- Modelware
- Language workbench
- Reverse eng. tool
- Benchmark
- Recommender
- Renovation tool
Parser [Aho et al]

- Syntactic analyser
- Traverses text
- Validates conformance to grammar
- Recognises language instances ("words")
- Constructs a parse tree

Parser [Lesk/Schmidt, Johnson]

- Two separate phases
  - lexical analysis (lexer)
  - syntactic analysis (parser)
- Write your grammar in a very specific style
- The parser is generated automatically
  - ...and maybe will work
- Want a good parser?
  - forget about readability and reuse

Parser [Cook et al]

- Recognises syntactic features
- Creates object structure
- Results in a graph
- Object grammars allow crosslinks
- Naturally bidirectional (parsing/rendering)
- Object grammars contain pretty-printing info

van der Storm, Cook, Loh, Object Grammars: Compositional & Bidirectional Mapping Between Text and Graphs, SLE 2012.
A mapping between
one of concrete syntaxes
the abstract syntax
Assumes the “spanning tree” is available or easy

Parser [Ophel]

- Parser has three tasks
  - verify that the input is valid
  - provide semantical framework
  - if invalid, explain why

To summarise

- Disregard textual parsing?
- Separate lexical from syntactic?
- Link to semantics?

http://commons.wikimedia.org/wiki/File:Torii_kiyoshige_bando_hikosaburo_ii.jpg
Despite the undeniable interest and importance of semantic and statistical studies of language, they appear to have no direct relevance to the problem of determining or characterizing the set of grammatical utterances. I think that we are forced to conclude that grammar is autonomous and independent of meaning.

Noam Chomsky
Semantics is a social contract #mpm13
#models2013
Grammars define structure & can assume different semantics
Joint work of:
Ali Afroozeh, Jeroen van den Bos, Magiel Bruntink, Jan van Eijck, Paul Griffioen, Mark Hills, Anastasia Izmaylova, Paul Klint, Wouter Kwakernaak, Dimitrios Kyritsis, Mattijn Lahuis, Davy Landman, Robert van Liere, Bert Lisser, George Marmanidis, Chris Mulder, Atze van der Ploeg, Riemer van Rozen, Alexander Serebrenik, Ashim Shahi, Sunil Simon, Michael Steindorfer, Tijs van der Storm, Ioannis Tzanellis, Jurgen Vinju, Kevin van der Vlist, Vadim Zaytsev (and others)
Language workbench

- Language oriented programming
- Little languages: grep, sed, awk
- Homoiconic languages: LISP, REBOL, XSLT
- Configuration files
- Other workbenches
  - MPS (Völter et al), Intentional Software (Simonyi et al), Epsilon (Paige et al)

Rascal Metaprogramming Language

Disclaimer
We currently only release alpha versions of Rascal, which are subject to frequent changes.

Eclipse plugin
The Eclipse update site for Rascal is: http://update.rascal-mpl.org/stable/

You need Eclipse for RCP and RAP Developers (Juno or Kepler) version of Eclipse available at www.eclipse.org/downloads/ to run Rascal. It has been reported recently that the latest release also works with normal (non RCP/RAP) versions of Eclipse. Not thoroughly tested though!

Please note that:
- Rascal now needs a JDK because it uses the Java compiler, so please download a JDK, not just a jRE.
- You may have to edit the Eclipse init file so that Eclipse can find your Java installation and Eclipse can allocate enough resources.
- For generating parsers, Rascal uses quite a bit of memory. Please use -vmargs -Xmx1G -Xss32m

Follow these steps to install the plugin into Eclipse
1. Start Eclipse.
2. Select Help -> Install New Software.
3. Make sure that the tick for "Contact all update sites during install to find required software" is enabled.
5. Select the feature Rascal.
6. Select Next (several times) and accept the software license. The process may take a few minutes!
7. Once these features have been installed, restart Eclipse.
Technical challenges

- How to parse source code/data files/models?
- How to extract facts from them?
- How to perform computations on these facts?
- How to generate new source code (transform, refactor, compile)?
- How to synthesize other information?

**EASY: Extract-Analyze-SYnthesize Paradigm**
Metaprogramming is EASY

- Extract
  - Fast context-free general top-down parsing
  - Pattern matching & generic traversal
- Analyze
  - Relational queries and comprehensions
  - Backtracking, fixed point computation, ...
- Synthesize
  - String templates
  - Concrete syntax
  - Interactive visualization generator
Metaprogramming is EASY

- **Extract**
  - Fast context-free general top-down parsing
  - Pattern matching & generic traversal

- **Analyze**
  - Relational queries and comprehensions
  - Backtracking, fixed point computation, ...

- **SYnthesize**
  - String templates
  - Concrete syntax
  - Interactive visualization generator
The EASY paradigm

System under investigation

Extract

Internal representation

Synthesize

Analyze

Results
Why a new language?

- No current technology spans the full range of EASY steps
- There are many fine technologies but they are
  - highly specialized with steep learning curves
  - hard to learn unintegrated technologies
  - not integrated with a standard IDE
  - hard to extend
- Goal: keep all benefits of ASF+SDF and Rscript
- in a new, unified, extensible, teachable framework
Bridging the gaps

Rascal Programming

Analysis
- Parsing/Matching
- Comprehension
- Projection
- Extraction
- Traversal

Data
- ASTs
- Sets
- relations

Figure

Visualization

Synthesis
- Abstract syntax
- Concrete syntax
- Rewriting
- Annotation
Rascal keywords

- Complex built-in data types
- Immutable data
- Static safety
- Generic types
- Local type inference
- Pattern matching
- Syntax definitions & parsing
- Concrete syntax
- Visiting/traversal
- Comprehensions
- Higher-order
- Familiar syntax
- Java and Eclipse integration
- Read-Eval-Print (REPL)
Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

presumably familiar
Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

installs as a plugin
Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

low barrier to entry, learn features as you go
Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

concrete syntax matching
Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

traversals, matching, …
Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

relations for sharing/merging of facts for different languages
To summarise

- Rascal is a language workbench
- EASY paradigm
  - extract
  - analyse
  - synthesise
- Metaprogramming
  - http://rascal-mpl.org

http://commons.wikimedia.org/wiki/File:Torii_kiyoshige_bando_hikosaburo_ii.jpg
Rascal features
Rich (immutable) data

- Built-in types:
  - lists
  - sets
  - maps
  - tuples
  - relations
  - with comprehensions and many operators

```
rascal> [1..10]
list[int]: [1,2,3,4,5,6,7,8,9,10]
```
```
rascal> [x/2 | x <- [1..10]]
list[int]: [0,1,1,2,2,3,3,4,4,5]
```
```
rascal> {x/2 | x <- [1..10]} + {4,5,6}
set[int]: {6,5,4,3,2,1,0}
```
Syntax definitions

- Define lexical syntax
- Define context-free syntax
- Define whitespace/layout/…
- Get GLL parser for free
- Define an algebraic data type
- Automatically implode parse trees to ASTs
Syntax definitions

lexical Id = [A-Za-züäöß]+ !>> [A-Za-züäöß];
lexical Num = [0-9]+ !>> [0-9];

- Define lexical syntax
- Define context-free syntax
- Define whitespace/layout/…
- Get GLL parser for free
- Define an algebraic data type
- Automatically implode parse trees to ASTs
Syntax definitions

- Define lexical syntax
- Define context-free syntax
- Define whitespace/layout/…
- Get GLL parser for free
- Define an algebraic data type
- Automatically implode parse trees to ASTs

```text
start syntax System = Line+;
syntax Line = Num "::" {Id ","}+ "." ;
```
Define lexical syntax
Define context-free syntax
Define whitespace/layout/…
Get GLL parser for free
Define an algebraic data type
Automatically implode parse trees to ASTs

```hs
layout WS = [	
]* !>> [	
];
```
Syntax definitions

- Define lexical syntax
- Define context-free syntax
- Define whitespace/layout/…
- Get GLL parser for free
- Define an algebraic data type
- Automatically implode parse trees to ASTs
Patterns

- Pattern matching
- on concrete syntax
- on lists
- on sets
- on trees
- ...
- Pattern-driven dispatch

```rascal
rascal> {int x, str y} := {2}
bool: false
rascal> {int x, str y} := {2,"3"}
bool: true
rascal> {int x, *y, str z} := {2,2,2,"3",4,"2"}
bool: true
```
Other pattern kinds

- **Regular**: grep/Perl like regular expressions
  
  `/^<before:\W*><word:\w+><after:.\*\$/`

- **Abstract**: match data types
  
  `whileStat(Exp, Stats*)`

- **Concrete**: match parse trees
  
  `` while <Exp> do <Stats*> od `
Pattern-directed invocation

```
bool eqfp(fpnt(), fpnt()) = true;
bool eqfp(fpopt(), fpopt()) = true;
bool eqfp(fpplus(), fpplus()) = true;
bool eqfp(fpstar(), fpstar()) = true;
bool eqfp(femptyy(), femptyy()) = true;
bool eqfp(fpmany(L1), fpmany(L2)) = multiseteq(L1,L2);
default bool eqfp(Footprint pi, Footprint xi) = false;
```

Zaytsev, Guided Grammar Convergence. Poster at SLE 2013
Switch/case

```java
switch(p)
{
    case (DCGFun)`[[]` => ["ε"];
    case (DCGFun)`<Word n>` =>
        ["<n>" | "<n>"==toLowerCase("<n>")];
    case (DCGFun)`(<<DCGFun ","}* args)` =>
        [*getTags(a) | a <- args];
    case (DCGFun)`<Word f> (<<DCGFun ","}* as)` =>
        ["<f>"] + [*getTags(a) | a <- as];
    default ... 
}
```
Visitor and foldr

@contributor{Bas Basten - Bas.Basten@cwi.nl (CWI)}
@contributor{Mark Hills - Mark.Hills@cwi.nl (CWI)}

module Operations

import AST;
import IO;

public Company cut(Company c) {
    return visit (c) {
        case employee(name, [*ep,ip:intProp("salary",salary),*ep2])
            => employee(name, [*ep,ip[intVal=salary/2],*ep2])
    }
}

public int total(Company c) {
    return (0 | it=salary | /employee(name, [*ep,ip:intProp
        ("salary",salary),*ep2]) <- c);
}
Polychromatic trees

data CTree = leaf(int N)
    | red(CTree left, CTree right)
    | black(CTree left, CTree right) ;

rb = red(black(leaf(1), red(leaf(2), leaf(3))), black(leaf(4), leaf(5))) ;

// TODO: count red nodes
Polychromatic trees

```java
data CTree = leaf(int N)
   | red(CTree left, CTree right)
   | black(CTree left, CTree right)
;

public int cntRed(CTree t) {
    switch(t){
        case leaf(_): return 0;
        case red(L,R):
            return 1 + cntRed(L) + cntRed(R);
        case black(L,R):
            return cntRed(L) + cntRed(R);
    }
}
```
Polychromatic trees

data CTree = leaf(int N)
  | red(CTree left, CTree right)
  | black(CTree left, CTree right) ;

public int cntRed(CTree t) {
  int c = 0;
  visit(t){case red(_,_): c += 1;};
  return c;
}
Polychromatic trees

data CTree = leaf(int N)  
|  red(CTree left, CTree right)  
|  black(CTree left, CTree right) ;

public int cntRed(CTree t) =  
size([n | /n:red(_,_) := t]);
public CTree frepl(CTree T) {
    return visit (T) {
        case red(CTree T1, Ctree T2) => green(T1, T2)
    };
}

public Ctree srepl(CTree T) {
    return top-down-break visit (T) {
        case red(Ctree T1, CTree T2) => green(T1, T2)
    };
}

public Ctree drepl(Ctree T) {
    return bottom-up-break visit (T) {
        case red(CTree T1, CTree T2) => green(T1, T2)
    };
}
To summarise

- Rascal is a language workbench
- Sets, relations, maps...
- Grammars, ADTs...
- Cool comprehensions
- Cool pattern matching
- Many more features
- http://rascal-mpl.org

http://commons.wikimedia.org/wiki/File:Torii_kiyoshige_bando_hikosaburo_ii.jpg
Grammar extraction
Grammar extraction

- Given is an artefact containing grammar knowledge:
  - a grammar
  - a parser specification
  - a metamodel
  - grammarware source code
  - a data schema
  - documentation
- How to extract a grammar from it?
Grammar extraction

- Annotation
- Grammar
- Metasyntax
- Grammar'
- Instance
- Recovery
- Extraction
- Inference
Variety of notations

Such details can be modeled

We compose a notation specification

Zaytsev, What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions, SAC/PL 2012.
Grammar notations: BNF

Arithmetic Expressions, Boolean Expressions, and Expressions

\[
\begin{align*}
\langle \text{factor} \rangle & \equiv \langle \text{number} \rangle \mid \langle \text{function} \rangle \mid \langle \text{variable} \rangle \mid \langle \text{subscr var} \rangle \mid (\langle \text{ar exp} \rangle) \mid \langle \text{factor} \rangle \langle \text{ar exp} \rangle \\
\langle \text{term} \rangle & \equiv \langle \text{factor} \rangle \langle \text{term} \rangle \langle \text{factor} \rangle \langle \text{term} \rangle \langle \text{factor} \rangle \langle \text{term} \rangle \\
\langle \text{ar exp} \rangle & \equiv \langle \text{term} \rangle \langle \text{factor} \rangle \langle \text{term} \rangle \langle \text{factor} \rangle \langle \text{term} \rangle \\
\langle \text{ar exp A} \rangle & \equiv \langle \text{ar exp} \rangle \\
\langle \text{relation} \rangle & \equiv \langle \text{ar exp} \rangle \langle \text{relation} \rangle \langle \text{ar exp A} \rangle \\
\langle \text{rel exp} \rangle & \equiv (\langle \text{ar exp} \rangle \langle \text{relation} \rangle \langle \text{ar exp A} \rangle) \\
\langle \text{bool term} \rangle & \equiv 0 \mid 1 \mid \langle \text{rel exp} \rangle \langle \text{function} \rangle \langle \text{variable} \rangle \langle \text{subscr var} \rangle \langle \text{bool exp} \rangle \\
\langle \text{bool exp} \rangle & \equiv \langle \text{bool term} \rangle \langle \text{bool exp} \rangle \lor \langle \text{bool term} \rangle \langle \text{bool exp} \rangle \land \langle \text{bool term} \rangle \langle \text{bool exp} \rangle \equiv \langle \text{bool term} \rangle \\
\langle \text{exp} \rangle & \equiv \langle \text{ar exp} \rangle \lor \langle \text{bool exp} \rangle
\end{align*}
\]
1 | Idents: 153
2 | - top: 1
3 | - used: 152
4 | - defined: 145
5 | - undefined: 8
6 | - Literals: 121
7 |
8 | ref-or-out
9 | : "ref"
10 | "out"
11 |
12 |
13 | expression-unary-operator
14 | : lex-csharp-extra/plus
15 | lex-csharp-extra/minus
16 | increment-decrement
17 | "!"
18 | "~"
19 | "x"
20 |
21 |
22 | increment-decrement
23 | : "++"
24 | "--"
25 |
26 |
27 | expression-shift-operator
28 | : "<<"
29 | ">>"
30 |
31 |
32 | expression-relational-operator
33 | : lex-csharp-extra/less-than
34 | lex-csharp-extra/greater-than
35 | "<"
36 | "<="
37 | ">="
38 | "ts"
39 |
40 |
41 | expression-equality-operator
42 | : "=="
43 | "!="
44 |
45 |
46 | conversion-kind
Metasyntactic evolution

Zaytsev, Language Evolution, Metasyntactically, EC-EASST 49 (BX), 2012.
EBNF in GrammarLab

```haskell
alias EBNF = map[MetasyMBOL,str];
data MetasyMBOL =
  start_grammar_symbol()
| end_grammar_symbol()
| epsilon_metasyMBOL()
| defining_symbol()
| multiple_defining_symbol()
| terminator_symbol()
| definition_separator_symbol()
| start_option_symbol()
| end_option_symbol()
| start_nonterminal_symbol()
| end_nonterminal_symbol()
...
```
public EBNF DefaultEBNF = {
    epsilon_metasyMBOL(): "ε",
defining_symbol(): " ::= ",
terminator_symbol(): " ;\n",
definition_separator_symbol(): " | ",
start_group_symbol(): "(",
end_group_symbol(): ")",
start_terminal_symbol(): "\"
end_terminal_symbol(): "\"
postfix_option_symbol(): "?",
...
};
IDE support

```
EBNF ebnf = ...;
edd2rsc(ebnf, |cwd:///EBNFGrammar.rsc|);

...

Tree getEBNF(str s, loc z)
    = parse(#EBNFGrammar,z);
public void registerEBNF()
    = registerLanguage("EBNF","bnf", getEBNF);
```
## Extraction of Java grammars

<table>
<thead>
<tr>
<th></th>
<th>impl1</th>
<th>impl2</th>
<th>impl3</th>
<th>read1</th>
<th>read2</th>
<th>read3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary lexical decisions</td>
<td>2</td>
<td>109</td>
<td>60</td>
<td>1</td>
<td>90</td>
<td>161</td>
<td>423</td>
</tr>
<tr>
<td>Well-formedness violations</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>4</td>
<td>31</td>
</tr>
<tr>
<td>Indentation violations</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>Recovery rules</td>
<td>3</td>
<td>12</td>
<td>18</td>
<td>2</td>
<td>59</td>
<td>47</td>
<td>141</td>
</tr>
<tr>
<td>○ Match parentheses</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>○ Metasymbol to terminal</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>27</td>
<td>7</td>
<td>42</td>
</tr>
<tr>
<td>○ Merge adjacent symbols</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>○ Split compound symbol</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>○ Nonterminal to terminal</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>11</td>
<td>29</td>
</tr>
<tr>
<td>○ Terminal to nonterminal</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>13</td>
<td>33</td>
</tr>
<tr>
<td>○ Recover optionality</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Purge duplicate definitions</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>51</td>
</tr>
<tr>
<td>Total</td>
<td>11</td>
<td>123</td>
<td>92</td>
<td>24</td>
<td>181</td>
<td>238</td>
<td>669</td>
</tr>
</tbody>
</table>

Extraction & recovery

EBNF ebnf = ...;

GGrammar bgf

= ebnf2bgf(ebnf, |cwd:///grammar.ebnf|);

GGrammar bgf

= hunter(ebnf, |cwd:///grammar.ebnf|);

// possibly with errors
Notation-parametric recovery

- Perform a robust heuristic-based recovery process
- Successful for grammars of Ada, C, C++, C#, Dart, Eiffel, Modula, MediaWiki, LLL, EBNF, etc.
Notation-parametric recovery

- Perform a robust heuristic-based recovery process
- Successful for grammars of Ada, C, C++, C#, Dart, Eiffel, Modula, MediaWiki, LLL, EBNF, etc.

Grammar extraction from other sources
Other available extractors

- Eclipse Modeling Framework
  
  \[
  \text{GGrammar bgf} = \text{ecore2bgf}(|\text{cwd:///metamodel.ecore}|); \\
  \]

- XML Schema Definition
  
  \[
  \text{GGrammar bgf} = \text{xsd2bgf}(|\text{cwd:///schema.xsd}|); \\
  \]

Not all extractors have already been migrated to use Rascal
Flashy megamodel!
Flashy megamodel!!!
Grammar Zoo

- 569 grammars
Grammar evolution
Grammar evolution

- Any change is a transformation
- Why transform?
  - adaptation
  - normalisation
  - beautification
  - inconsistency management
  - version control
- Documented, well-understood, compositional change
- Good for representing relationships
Grammar evolution

Annotation  Grammar  Grammar'  Instance

Metasyntax
Transformation components
Transformation components

- known semantics, well-defined algorithm
- rename, fold, factor, inject, remove, …
Transformation components

Arguments

• what exactly to rename/factor/inject/…?
Transformation components

Input grammar

- determines applicability
Transformation components

Output grammar

- inferred result
### Excerpt from the operator suite

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bypass()</td>
<td>preserving</td>
<td>updative</td>
<td>not breaking</td>
</tr>
<tr>
<td>factor$(x, y)$</td>
<td>preserving</td>
<td>updative</td>
<td>not breaking</td>
</tr>
<tr>
<td>introduce$(n ::= rhs)$</td>
<td>preserving</td>
<td>additive</td>
<td>not breaking</td>
</tr>
<tr>
<td>eliminate$(n)$</td>
<td>preserving</td>
<td>subtractive</td>
<td>not breaking</td>
</tr>
<tr>
<td>widen$(x, y)$</td>
<td>increasing</td>
<td>additive</td>
<td>not breaking</td>
</tr>
<tr>
<td>define$(n ::= rhs)$</td>
<td>revising</td>
<td>additive</td>
<td>not breaking</td>
</tr>
<tr>
<td>renameN$(a, b)$</td>
<td>preserving</td>
<td>additive</td>
<td>resolvable</td>
</tr>
<tr>
<td>fold$(n)$</td>
<td>preserving</td>
<td>additive</td>
<td>resolvable</td>
</tr>
<tr>
<td>extract$(n ::= rhs)$</td>
<td>preserving</td>
<td>additive</td>
<td>resolvable</td>
</tr>
<tr>
<td>unfold$(n)$</td>
<td>preserving</td>
<td>subtractive</td>
<td>resolvable</td>
</tr>
<tr>
<td>disappear$(p, m)$</td>
<td>decreasing</td>
<td>subtractive</td>
<td>resolvable</td>
</tr>
<tr>
<td>permute$(p, q)$</td>
<td>revising</td>
<td>updative</td>
<td>resolvable</td>
</tr>
<tr>
<td>concretize$(p, m)$</td>
<td>revising</td>
<td>additive</td>
<td>resolvable</td>
</tr>
<tr>
<td>abstractize$(p, m)$</td>
<td>revising</td>
<td>subtractive</td>
<td>resolvable</td>
</tr>
<tr>
<td>project$(p, m)$</td>
<td>revising</td>
<td>subtractive</td>
<td>resolvable</td>
</tr>
<tr>
<td>narrow$(x, y)$</td>
<td>decreasing</td>
<td>subtractive</td>
<td>unresolvable</td>
</tr>
<tr>
<td>inject$(p, m)$</td>
<td>revising</td>
<td>additive</td>
<td>unresolvable</td>
</tr>
<tr>
<td>replace$(x, y)$</td>
<td>revising</td>
<td>—</td>
<td>unresolvable</td>
</tr>
</tbody>
</table>

---

**Table 1** – A representative excerpt from the XBGF operator suite. Among operands $a, b, n$ are nonterminals, $p, q$ are production rules, $x, y$ are grammatical expressions, $m$ are markers. A “preserving” transformation preserves the language defined by the grammar, an “increasing” or “decreasing” one makes it larger or smaller, and a “revising” operator can have other outcomes. An “updative” transformation preserves the size of the modelling space, an “additive” or “subtractive” transformation makes it larger or smaller. A “not breaking” metamodel transformation does not invalidate language instances, a “resolvable” one implies an algorithm for automatic coevolution, and an “unresolvable” requires manual work or extra data.

XBGF has many more operators and finds its uses in grammar recovery for correcting mistakes in the original grammar, containing artefacts or ones introduced by the automated extraction process, grammar specialisation for automatically deriving a tool-specific grammar from a baseline grammar, grammar convergence for validating claims about language or grammar equivalence, grammar beautification for increasing readability of a grammar, technological space travel for transforming a grammar in a narrow sense to a database schema or a class diagram to an algebraic data type, etc.

Many authors — in particular, Herrmannsdörfer et al [HBJ79], [HVWtt], Cicchetti et al [CREP78], Wachsmuth [Wac70] — have previously considered or analysed metamodel transformation operators that are strikingly similar to grammar transformation operators. In this paper, we have limited ourselves to grammar transformation not only because grammars are considered somewhat simpler than other types of models, but also because there are many more operators that are specific to grammars.

---

Zaytsev, *Negotiated Grammar Transformation*, JOT. (under review)
Table 6: Transformation of the JLS grammars — effort metrics and categorization

<table>
<thead>
<tr>
<th></th>
<th>jls1</th>
<th>jls12</th>
<th>jls123</th>
<th>jls2</th>
<th>jls3</th>
<th>read12</th>
<th>read123</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of lines</td>
<td>682</td>
<td>5114</td>
<td>2847</td>
<td>6774</td>
<td>10721</td>
<td>1639</td>
<td>3082</td>
<td>30859</td>
</tr>
<tr>
<td>Number of transformations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Semantics-preserving</td>
<td>67</td>
<td>290</td>
<td>111</td>
<td>387</td>
<td>544</td>
<td>77</td>
<td>135</td>
<td>1611</td>
</tr>
<tr>
<td>- Semantics-increasing/-decreasing</td>
<td>45</td>
<td>231</td>
<td>80</td>
<td>275</td>
<td>381</td>
<td>31</td>
<td>78</td>
<td>1121</td>
</tr>
<tr>
<td>- Semantics-revising</td>
<td>22</td>
<td>58</td>
<td>31</td>
<td>102</td>
<td>150</td>
<td>39</td>
<td>53</td>
<td>455</td>
</tr>
<tr>
<td>Preparation phase (§4.2.1)</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>15</td>
<td>24</td>
<td>11</td>
<td>14</td>
<td>65</td>
</tr>
<tr>
<td>- Known bugs</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>11</td>
<td>—</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>- Post-extraction</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>27</td>
</tr>
<tr>
<td>- Initial correction</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>Resolution phase</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Extension (§4.2.3)</td>
<td>21</td>
<td>59</td>
<td>31</td>
<td>97</td>
<td>139</td>
<td>35</td>
<td>43</td>
<td>425</td>
</tr>
<tr>
<td>- Relaxation (§4.2.4)</td>
<td>—</td>
<td>17</td>
<td>26</td>
<td>—</td>
<td>—</td>
<td>31</td>
<td>38</td>
<td>112</td>
</tr>
<tr>
<td>- Correction (§4.2.5)</td>
<td>18</td>
<td>39</td>
<td>5</td>
<td>75</td>
<td>112</td>
<td>—</td>
<td>2</td>
<td>251</td>
</tr>
</tbody>
</table>

Grammar programming
Grammar programming

unfold(n)
Meanwhile in modelware...

Model Migration Approaches:

Operator-based co-evolution

+ migration strategy is “free”
- tool lock in
- operator set completeness
- reverse engineering challenging
A transformation sequence

expr ::= ...;
atom ::= ID | INT | '(' expr ')';

bool case_deyaccify_left_plus (bool debug)
{
    GGrammar bgf = grammar(["foo"],
        [production("foo",
          nonterminal("bar")),
        production("foo",
            sequence([
            nonterminal("foo"),
            nonterminal("bar")]))],
        []);

    XSequence xbgf =
        [deyaccify("foo")];

    GGrammar bl = grammar(["foo"],
        [production("foo",
            plus(nonterminal("bar"))],
        []);

    return run_case(bgf, xbgf, bl, debug);
}

test bool test_deyaccify_left_plus() = case_deyaccify_left_plus(false);
void show_deyaccify_left_plus() {case_deyaccify_left_plus(true);}
bool case_deyaccify_left_plus(bool debug)
{
    GGrammar bgf = grammar(["foo"],
        [production("foo",
                    nonterminal("bar")),
            production("foo",
                        sequence([nonterminal("foo"),
                                    nonterminal("bar")]))],
        []);
    XSequence xbgf =
        [deyaccify("foo")];
    GGrammar bl = grammar(["foo"],
        [production("foo",
                    plus(nonterminal("bar")))]);
    []);
    return run_case(bgf, xbgf, bl, debug);

rascal> import tests::transform::XBGF;
ok

rascal>:test
ok

rascal>
Negotiated transformations
Rename a nonterminal: success

rename(expr, Expr)

ok
Rename a nonterminal: failure

rename(expr, Expr)

no expr!
Rename a nonterminal: negotiation

rename(expr, Expr)

no expr!

rename(exp, Exp)

ok

Zaytsev, Negotiated Grammar Transformation, XM2012 @ MODELS 2012. JOT.
Rename a nonterminal: negotiation

\texttt{rename(expr,Expr)}

no expr!

\texttt{rename(exp,Exp)}

ok

Zaytsev, \textit{Negotiated Grammar Transformation}, XM2012 @ MODELS 2012. JOT.
Pending evolution

KEEP CALM AND CONSIDER IT DONE

Pending evolution
Grammar programming
Optional execution

Zaytsev, Pending Evolution of Grammars, XM2013 @ MODELS 2013.
Asserting preconditions

\[ G \rightarrow G \rightarrow G \rightarrow G \rightarrow G \]

vertical
deyaccify
deyaccify
vertical

Zaytsev, Pending Evolution of Grammars, XM2013 @ MODELS 2013.
Reuse correction scripts

---

inject

defineN

deyaccify

replace

Zaytsev, Pending Evolution of Grammars, XM2013 @ MODELS 2013.
Reuse correction scripts

$G \xrightarrow{\text{injection}} G \xrightarrow{\text{defineN}} G \xrightarrow{\text{deyaccify}} G \xrightarrow{\text{replace}} G \xrightarrow{\text{report}}$
Normalise before export

Zaytsev, Pending Evolution of Grammars, XM2013 @ MODELS 2013.
Operator-based evolution analysis
Table 7: XBGF operators usage for JLS convergence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>jls1</th>
<th>jls12</th>
<th>jls123</th>
<th>jls2</th>
<th>jls3</th>
<th>read12</th>
<th>read123</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>rename</td>
<td>9</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>10</td>
<td>—</td>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>reread</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>unfold</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>11</td>
<td>13</td>
<td>2</td>
<td>3</td>
<td>48</td>
</tr>
<tr>
<td>fold</td>
<td>4</td>
<td>11</td>
<td>4</td>
<td>11</td>
<td>13</td>
<td>2</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>inline</td>
<td>3</td>
<td>67</td>
<td>8</td>
<td>71</td>
<td>100</td>
<td>—</td>
<td>1</td>
<td>250</td>
</tr>
<tr>
<td>extract</td>
<td>—</td>
<td>17</td>
<td>5</td>
<td>18</td>
<td>30</td>
<td>—</td>
<td>5</td>
<td>75</td>
</tr>
<tr>
<td>chain</td>
<td>1</td>
<td>—</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>massage</td>
<td>2</td>
<td>13</td>
<td>—</td>
<td>15</td>
<td>32</td>
<td>5</td>
<td>3</td>
<td>70</td>
</tr>
<tr>
<td>distribute</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>—</td>
<td>—</td>
<td>18</td>
</tr>
<tr>
<td>factor</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>24</td>
<td>3</td>
<td>1</td>
<td>44</td>
</tr>
<tr>
<td>deyaccify</td>
<td>2</td>
<td>20</td>
<td>—</td>
<td>25</td>
<td>33</td>
<td>4</td>
<td>3</td>
<td>87</td>
</tr>
<tr>
<td>yaccify</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>eliminate</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>14</td>
<td>22</td>
<td>—</td>
<td>—</td>
<td>46</td>
</tr>
<tr>
<td>introduce</td>
<td>—</td>
<td>1</td>
<td>30</td>
<td>4</td>
<td>13</td>
<td>3</td>
<td>34</td>
<td>85</td>
</tr>
<tr>
<td>import</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
</tr>
<tr>
<td>vertical</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>22</td>
<td>5</td>
<td>8</td>
<td>62</td>
</tr>
<tr>
<td>horizontal</td>
<td>4</td>
<td>19</td>
<td>5</td>
<td>17</td>
<td>31</td>
<td>4</td>
<td>4</td>
<td>84</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>14</td>
<td>13</td>
<td>7</td>
<td>20</td>
<td>28</td>
<td>20</td>
<td>103</td>
</tr>
<tr>
<td>appear</td>
<td>—</td>
<td>8</td>
<td>11</td>
<td>8</td>
<td>25</td>
<td>2</td>
<td>17</td>
<td>71</td>
</tr>
<tr>
<td>widen</td>
<td>1</td>
<td>3</td>
<td>—</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>upgrade</td>
<td>—</td>
<td>8</td>
<td>—</td>
<td>14</td>
<td>20</td>
<td>2</td>
<td>2</td>
<td>46</td>
</tr>
<tr>
<td>unite</td>
<td>18</td>
<td>2</td>
<td>—</td>
<td>18</td>
<td>21</td>
<td>5</td>
<td>4</td>
<td>68</td>
</tr>
<tr>
<td>remove</td>
<td>—</td>
<td>10</td>
<td>1</td>
<td>11</td>
<td>18</td>
<td>—</td>
<td>1</td>
<td>41</td>
</tr>
<tr>
<td>disappear</td>
<td>—</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>—</td>
<td>—</td>
<td>33</td>
</tr>
<tr>
<td>narrow</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>4</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>downgrade</td>
<td>—</td>
<td>2</td>
<td>—</td>
<td>8</td>
<td>3</td>
<td>—</td>
<td>—</td>
<td>13</td>
</tr>
<tr>
<td>define</td>
<td>—</td>
<td>6</td>
<td>—</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>6</td>
<td>26</td>
</tr>
<tr>
<td>undefined</td>
<td>—</td>
<td>3</td>
<td>—</td>
<td>5</td>
<td>3</td>
<td>—</td>
<td>—</td>
<td>11</td>
</tr>
<tr>
<td>redefine</td>
<td>—</td>
<td>3</td>
<td>—</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td>inject</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>4</td>
<td>—</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>project</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>4</td>
</tr>
<tr>
<td>replace</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>unlabel</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Grammar testing
Grammar-based testing

- Asymmetric comparison:
  - Reference grammar vs. parser under test
- Symmetric comparison:
  - Differential testing
  - Systematic test data generation
  - Controlled combinatorial coverage
  - Larger sets of smaller test data items
  - Nonterminal matching
  - Non-context-free effects
Coverage criteria

- **Trivial coverage**: if the test data set is not empty.
- **Nonterminal coverage**: if each nonterminal is exercised at least once.
- **Production coverage**: if each production rule in the grammar is exercised at least once.
- **Branch coverage**: each branch of `? | *+`
- **Unfolding coverage**: each production of each right hand side nonterminal occurrence
- **Context-dependent branch coverage**!
Nonterminal matching
Nonterminal matching

<table>
<thead>
<tr>
<th></th>
<th>block</th>
<th>nearly perfect</th>
<th>universal</th>
<th>exclusive</th>
<th>void</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exclusive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nearly perfect</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>universal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Nonterminal matching
Nonterminal matching
Language documentation
Language documentation

- Given are:
  - a grammar for a software language
  - explanations in a natural language
  - executable code samples
  - known relationships between concepts

- How to do language documentation properly?
- By generating it from structured data!
Language documentation

Metasyntax

Annotation ↔ Grammar

Grammar ↔ Grammar’

Instance
Language Standardization Needs Grammarware

Steven Klusener, Vadim Zaytsev
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Email: {steven, vadim}@cs.vu.nl

October 4, 2005

Abstract

The ISO programming language standards are valuable documents that describe the syntax and semantics of mainstream languages. New features are proposed after thorough reviews by the standardization committees, leading to change documents that describe which modifications have to be enforced in the language standard document in order to actually add a new feature to the language. Maintaining these documents, both the language standard itself and all the change documents, is a time and resource consuming effort and in the evolution of these documents inconsistencies may be introduced. In this note we propose to utilize grammarware, a collection of new methods and new technology which can be used to support the advancement of these language documents in a more structured way. Besides, we will discuss how other tooling (like browsable language definitions, parser generators, pretty-printers, code checkers, etc.) can be obtained from the language standard. The final objective is threefold: (1) to facilitate the standardization committees in their activities and to raise the quality of the language standard documents; (2) to extend the usability of language standards by providing various presentations of each standard (in a human readable document, in a browsable form, in a machine readable BNF, etc.); (3) to help tool builders (compiler vendors, IDE vendors, etc.) in generating their parsing front-end, and to provide technology for tool builders to specify differences between their dialects and the actual standard.

1 Introduction

Software engineering is an established area, both in the IT industry and in the academic field of computer science. In the recent years a lot of progress within software engineering is made in (among others):

1. Computer languages and specification formalisms, with a continuous increase in the level of abstraction and declarativity;

2. Tool support. Software is not written in simple text editors anymore, but in advanced Integrated Development Environments (IDEs), like Microsoft Visual Studio, IBM WebSphere Studio and Eclipse, Borland Enterprise Studio, Anjuta DevStudio, etc.;
### Unified model for language docs

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>synopsis</td>
<td>~</td>
<td>~</td>
<td>intent</td>
<td>synopsis</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>description</td>
<td>~</td>
<td>~</td>
<td>motivation</td>
<td>definition</td>
<td>usage</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>syntax</td>
<td>~</td>
<td>~</td>
<td>structure</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>constraints</td>
<td>~ a</td>
<td>syntax</td>
<td>structure</td>
<td>errors</td>
<td>restrictions</td>
<td>~</td>
<td>~</td>
<td>constraints</td>
</tr>
<tr>
<td>references</td>
<td>~</td>
<td>constraints</td>
<td>related patterns</td>
<td>references</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>relationship</td>
<td>~</td>
<td>~</td>
<td>consequences</td>
<td>return value, refinement</td>
<td>related</td>
<td>return type</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>semantics</td>
<td>~</td>
<td>semantics</td>
<td>collaborations</td>
<td>important</td>
<td>~</td>
<td>semantics</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>rationale</td>
<td>~</td>
<td>notes</td>
<td>implementation</td>
<td>rationale</td>
<td>GLS, ES c</td>
<td>note</td>
<td>rationale</td>
<td>note</td>
</tr>
<tr>
<td>example</td>
<td>examples</td>
<td>examples</td>
<td>sample code, known uses</td>
<td>~</td>
<td>~</td>
<td>example</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>update</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>changes</td>
<td>~</td>
</tr>
<tr>
<td>default</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>note</td>
<td>default values</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>value</td>
<td>~</td>
<td>~</td>
<td>also known as</td>
<td>conforms to</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>list</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>messages, parameters</td>
<td>terminals</td>
<td>~</td>
<td>properties</td>
<td>~</td>
</tr>
<tr>
<td>section</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td>subtopic</td>
<td>~</td>
<td>types</td>
<td>participants</td>
<td>fields</td>
<td>parameters, methods</td>
<td>~</td>
<td>operations</td>
<td>functions</td>
</tr>
<tr>
<td>Coverage of LDF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grammar
export
module CSharp

import ParseTree;
import util::IDE;
import IO;

layout WS = [\t-\n\r]* !>> [\t-\n\r];

start syntax CompilationUnit = UsingDirectives? GlobalAttributes? NamespaceMemberDeclarations?;

syntax UsingDirectives = UsingDirective | (UsingDirectives UsingDirective);


syntax UsingAliasDirective = "using" Identifier "=" NamespaceOrTypeName ";";

...
Generated SDF

module Main

exports
context-free start-symbols Compilation-unit
sorts
...

context-free syntax

   Using-directive → Using-directives
   Using-directives Using-directive → Using-directives

   Using-alias-directive → Using-directive
   Using-namespace-directive → Using-directive

   "using" Identifier "=" Namespace-or-type-name ";" → Using-alias-directive

...
# Browsable C# 1.x Grammar

**Summary**

- Number of production rules: 240
- Number of top alternatives: 584
- Number of defined nonterminal symbols: 240
- Root nonterminal symbols: —
- Other top nonterminal symbols: 1: `compilation-unit`
- Bottom nonterminal symbols: 2: `identifier`, `literal`
- Number of used terminal symbols: 126
- Special terminal symbols: 44: `.`, `[,]`, `:`, `;`, `(`, `)`
- Keywords: 82: `bool`, `decimal`, `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `object`, `string`, `ref`, `out`, `this`, `base`, `new`, `typeof`, `void`, `checked`, `unchecked`, `is`, `as`, `const`, `if`, `else`, `switch`, `case`, `default`, `while`, `do`, `for`, `foreach`, `in`, `break`, `continue`, `goto`, `return`, `throw`, `try`, `catch`, `finally`, `lock`, `using`, `namespace`, `class`, `public`, `protected`, `internal`, `private`, `abstract`, `sealed`, `static`, `readonly`, `volatile`, `virtual`, `override`, `extern`, `params`, `get`, `set`, `event`, `add`, `remove`, `operator`, `true`, `false`, `implicit`, `explicit`, `struct`, `interface`, `enum`, `delegate`, `assembly`, `field`, `method`, `module`, `param`, `property`

**Syntax**

```plaintext
namespace-name:
  namespace-or-type-name

type-name:
  namespace-or-type-name
```

Grammar analysis
Grammar-based metrics

- PROD, VAR, TERM, ...
- VOC, LEN, EFF, PLEV, LLEV, ...
- LEV, CLEV, RLEV, NLEV, ...
- cf.
Micropatterns

- Recognisable
- Purposeful
- Prevalent
- Simple
- Scoped
- Observed

Zaytsev, *Micropatterns in Grammars*. SLE 2013
To summarise

- Extraction
- recovery
- Evolution
- transformation
- negotiated
- pending
- Documentation
- Export

http://commons.wikimedia.org/wiki/File:Torii_kiyoshige_bando_hikosaburo_ii.jpg
Questions?
(What else would you like?)

vadim@grammarware.net