Guided Grammar Convergence
Full Case Study Report
Generated by converge::Guided

Vadim Zaytsev, vadim@grammarware.net,
Software Analysis & Transformation Team (SWAT),
Centrum Wiskunde & Informatica (CWI), The Netherlands

May 2, 2014
Introduction

This report is meant to be used as auxiliary material for the guided grammar convergence technique proposed in [18] as problem-specific improvement on [12]. It contains a megamodel renarrated as proposed in [19], as well as full results of the guided grammar convergence experiment on the Factorial Language, with details about each grammar source packaged in a readable form. All formulae used within this document, are generated automatically by the convergence infrastructure in order to avoid any mistakes. The generator source code and the source of the introduction text can be found publicly available in the Software Language Processing Suite repository [21].

Consider the model on Figure 1. It is a megamodel in the sense of [1, 6], since it depicts a linguistic architecture: all nodes represent software languages and language transformations, and all edges represent relationships between them. MegaL [5] is used as a notation: blue boxes represent tangible artefacts (files, programs, modules, directories, collections of other concrete entities), yellow boxes denote software languages in the broad sense (from general purpose languages to domain specific ones), and green boxes represent transformations (such as transformations applied to the model, or transformations of languages).

[Figure 1: Guided grammar convergence megamodel.]
programming languages to data types and protocols), light green boxes are used for functions (in fact, model transformations) and dark green boxes are for function applications.

As we can see from Figure 1 if we start reading it from the bottom, there is a program Guided.rsc, which was written in Rascal metaprogramming language [11]. It implements the guided grammar convergence process, which input language is BGF (BNF-like Grammar Formalism, a straightforward internal representation format for grammars, introduced in [12]). Its output language is ΞBGF, a bidirectional grammar transformation language introduced in [20]. An application of the guided grammar convergence algorithm to two grammars: one master grammar defining the intended software language (terminology of [18]) and one servant grammar (its label displayed in italics since it is actually a variable, not a concrete entity) — yields a transformation script that implements a grammar transformation than indeed transforms the servant grammar into the master grammar. The process behind this inference is relatively complicated and involves triggered grammar design mutations, normalisation to Abstract Normal Form, constructing weak prodsig-equivalence (≎) classes and resolving nominal and structural differences, as described on the theoretic level in [18].

The rest of the report presents instantiations of this megamodel for eleven concrete grammar sources:

- **adt**: an algebraic data type\(^1\) in Rascal [10];
- **antlr**: a parser description in the input language of ANTLR [15]. Semantic actions (in Java) are intertwined with EBNF-like productions.
- **dcg**: a logic program written in the style of definite clause grammars [16].
- **emf**: an Ecore model [14], automatically generated by Eclipse [3] from the XML Schema of the xsd source;
- **jaxb**: an object model obtained by a data binding framework. Generated automatically by JAXB [7] from the XML schema for FL.
- **om**: a hand-crafted object model (Java classes) for the abstract syntax of FL.
- **python**: a parser specification in a scripting language, using the PyParsing library [13];
- **rascal**: a concrete syntax specification in Rascal metaprogramming language [10, 11];
- **sdf**: a concrete syntax definition in the notation of SDF [9] with scannerless generalized LR [4, 17] as a parsing model.
- **txl**: a concrete syntax definition in the notation of TXL (Turing eXtender Language) transformational framework [2], which, unlike SDF, uses a combination of pattern matching and term rewriting).
- **xsd**: an XML schema [8] for the abstract syntax of FL.

\(^1\)http://tutor.rascal-mp1.org/Courses/Rascal/Declarations/AlgebraicDataType/AlgebraicDataType.html.
## Contents

1 ANTLR .......................... 5
   1.1 Source grammar .................. 5
   1.2 Mutations ........................ 5
   1.3 Normalizations .................... 6
   1.4 Grammar in ANF ................... 7
   1.5 Nominal resolution ............... 7
   1.6 Structural resolution .......... 8

2 Definite Clause Grammar .......... 9
   2.1 Source grammar ................... 9
   2.2 Mutations ........................ 9
   2.3 Normalizations .................... 9
   2.4 Grammar in ANF ................... 10
   2.5 Nominal resolution ............... 11
   2.6 Structural resolution .......... 11

3 Eclipse Modeling Framework ....... 12
   3.1 Source grammar ................... 12
   3.2 Normalizations .................... 12
   3.3 Grammar in ANF ................... 13
   3.4 Nominal resolution ............... 14
   3.5 Structural resolution .......... 14

4 JAXB Data Binding Framework .... 15
   4.1 Source grammar ................... 15
   4.2 Normalizations .................... 16
   4.3 Grammar in ANF ................... 17
   4.4 Nominal resolution ............... 17
   4.5 Structural resolution .......... 17

5 Java Object Model ................ 19
   5.1 Source grammar ................... 19
   5.2 Normalizations .................... 19
   5.3 Grammar in ANF ................... 21
   5.4 Nominal resolution ............... 21
   5.5 Structural resolution .......... 21
| Section | Title | Page |
|---------|-------|------|
| 6       | PyParsing in Python | 23   |
| 6.1     | Source grammar       | 23   |
| 6.2     | Mutations            | 23   |
| 6.3     | Normalizations       | 24   |
| 6.4     | Grammar in ANF       | 25   |
| 6.5     | Nominal resolution   | 25   |
| 6.6     | Structural resolution| 26   |
| 7       | Rascal Algebraic Data Type | 27 |
| 7.1     | Source grammar       | 27   |
| 7.2     | Normalizations       | 27   |
| 7.3     | Grammar in ANF       | 28   |
| 7.4     | Nominal resolution   | 28   |
| 7.5     | Structural resolution| 29   |
| 8       | Rascal Concrete Syntax Definition | 30 |
| 8.1     | Source grammar       | 30   |
| 8.2     | Normalizations       | 30   |
| 8.3     | Grammar in ANF       | 32   |
| 8.4     | Nominal resolution   | 32   |
| 9       | Syntax Definition Formalism | 33 |
| 9.1     | Source grammar       | 33   |
| 9.2     | Normalizations       | 33   |
| 9.3     | Grammar in ANF       | 34   |
| 9.4     | Nominal resolution   | 35   |
| 9.5     | Structural resolution| 35   |
| 10      | TXL                 | 36   |
| 10.1    | Source grammar       | 36   |
| 10.2    | Normalizations       | 36   |
| 10.3    | Grammar in ANF       | 37   |
| 10.4    | Nominal resolution   | 37   |
| 10.5    | Structural resolution| 38   |
| 11      | XML Schema           | 39   |
| 11.1    | Source grammar       | 39   |
| 11.2    | Normalizations       | 39   |
| 11.3    | Grammar in ANF       | 40   |
| 11.4    | Nominal resolution   | 41   |
| 11.5    | Structural resolution| 41   |
Grammar 1

ANTLR

Source name: antlr

1.1 Source grammar

- Source artifact: topics/fl/java1/FL.g
- Grammar extractor: topics/extraction/antlr/antlrstrip.py
- Grammar extractor: topics/extraction/antlr/slps/antlr2bgf/StrippedANTLR.g

| Production rules |
|------------------|
| p('', program, +(sel ('f', function))) |
| p('', function, seq ((sel ('a', ID), +(sel ('a', ID)), ':='), sel ('e', expr), +(NEWLINE))) |
| p('', expr, choice((sel ('b', binary), sel ('a', apply)), sel ('i', ifThenElse))) |
| p('', binary, seq ([sel ('l', atom), ∗(seq ([sel ('o', ops), sel ('r', expr)]))]) |
| p('', apply, seq ([sel ('i', ID), +(sel ('a', atom))]) |
| p('', ifThenElse, seq ([['if', sel ('c', expr), 'then', sel ('e1', expr), 'else', sel ('e2', expr)])]) |
| p('', atom, choice([ID, INT], seq ([['(', sel ('e', expr), ')']])]) |
| p('', ops, choice([['==', '+', '-']])) |

1.2 Mutations

- unite-splitN expr
  p ('', atom, choice([ID, INT, seq ([['(', sel ('e', expr), ')']])]) |
- designate-unlabel
  p ('tmplabel', binary, seq ([sel ('l', expr), +(seq ([sel ('a', ops), sel ('r', expr)])])]) |
- anonymize-deanonymize
  p ('tmplabel', binary, seq ([sel ('l', expr), +(seq ([sel ('a', ops), sel ('r', expr)])])]) |
- assoc-iterate
  p ('tmplabel', binary, seq ([sel ('l', expr), sel ('o', ops), sel ('r', expr)])) |
- deanonymize-anonymize
  p ('tmplabel', binary, seq ([sel ('l', expr), sel ('o', ops), sel ('r', expr)])) |
- unlabel-designate
  p ('tmplabel', binary, seq ([sel ('l', expr), sel ('a', ops), sel ('r', expr)])) |
1.3 Normalizations

- **reroor-reroot** [] to \([\text{program}]\)
- **anonymize-deanonymize**
  \(p\left(\text{', function, seq}\left(\text{sel}('a', ID), '='; \text{sel}('e', \text{expr}) + \text{(NEWLINE)}\right)\right)\)
- **anonymize-deanonymize**
  \(p\left(\text{'', program, +}\left(\text{sel}(F, \text{function})\right)\right)\)
- **anonymize-deanonymize**
  \(p\left(\text{'', ifThenElse, seq}\left(['if', \text{sel}('c', \\text{expr}) 'then', \text{sel}('e1', \text{expr}) 'else', \text{sel}('e2', \text{expr})]\right)\right)\)
- **anonymize-deanonymize**
  \(p\left(\text{'', binary, seq}\left(\text{sel}('l', \text{expr}), \text{sel}('o', \text{ops}), \text{sel}('r', \text{expr})\right)\right)\)
- **anonymize-deanonymize**
  \(p\left(\text{'', expr, choice}\left(\text{ID, INT, seq}\left(['(', \text{expr}, ')']\right)\right)\right)\)
- **abstractize-concretize**
  \(p\left(\text{'', ops, choice}\left([==, +, -]\right)\right)\)
- **abstractize-concretize**
  \(p\left(\text{'', expr, choice}\left([ID, INT, seq]\left(['(', \text{expr}, ')']\right)\right)\right)\)
- **abstractize-concretize**
  \(p\left(\text{'', function, seq}\left([ID, +(ID), \text{expr}, +(\text{NEWLINE})]\right)\right)\)
- **abstractize-concretize**
  \(p\left(\text{'', ifThenElse, seq}\left([ID, \text{expr}, \text{then expr, else expr}]\right)\right)\)
- **vertical-horizontal** in \(\text{expr}\)
- **undefined-define**
  \(p\left(\text{'', ops, e}\right)\)
- **unchain-chain**
  \(p\left(\text{'', expr, binary}\right)\)
- **unchain-chain**
  \(p\left(\text{'', expr, apply}\right)\)
- **unchain-chain**
  \(p\left(\text{'', expr, ifThenElse}\right)\)
- **abridge-detour**
  \(p\left(\text{'', expr, expr}\right)\)
- **unlabel-designate**
  \(p\left(\text{'', binary, expr, seq}\left(\text{expr, ops, expr}\right)\right)\)
- **unlabel-designate**
  \(p\left(\text{'', apply, expr, seq}\left([ID, +(expr)]\right)\right)\)
- **unlabel-designate**
  \(p\left(\text{'', ifThenElse, expr, seq}\left([expr, expr, expr]\right)\right)\)
- **extract-inline** in \(\text{expr}\)
  \(p\left(\text{'', expr1, seq}\left([expr, ops, expr]\right)\right)\)
- **extract-inline** in \(\text{expr}\)
  \(p\left(\text{'', expr2, seq}\left([ID, +(expr)]\right)\right)\)
- **extract-inline** in \(\text{expr}\)
  \(p\left(\text{'', expr3, seq}\left([expr, expr, expr]\right)\right)\)
1.4 Grammar in ANF

| Production rule | Production signature |
|----------------|----------------------|
| p(‘c, program, +(function)) | {<function, +>} |
| p(‘c, function, seq([ID, -(ID), expr, +(NEWLINE)])) | {<expr, 1>, (NEWLINE, +), (ID, 1+)} |
| p(‘c, expr, ID) | {<ID, 1>}, INT | |
| p(‘c, expr, INT) | {<expr, 1>} |
| p(‘c, expr, expr1) | {<expr2, 1>} |
| p(‘c, expr, expr2) | {<expr2, 1>} |
| p(‘c, expr, expr3) | {<expr3, 1>} |
| p(‘c, expr1, seq([expr, ops, expr])) | {<ops, 1>, (expr, 11)} |
| p(‘c, expr2, seq([ID, +(expr)])) | {<expr+, (ID, 1)}> |
| p(‘c, expr3, seq([expr, expr, expr])) | {<expr111>} |

1.5 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

- `p(‘c, program, +(function)) = p(‘c, program, +(function))`
- `p(‘c, function, seq([ID, -(ID), expr, +(NEWLINE)])) = p(‘c, function, seq([str, +(str), expression]))`
- `p(‘c, expr, ID) = p(‘c, expression, str)`
- `p(‘c, expr, INT) = p(‘c, expression, int)`
- `p(‘c, expr, expr1) = p(‘c, expression, binary)`
- `p(‘c, expr, expr2) = p(‘c, expression, apply)`
- `p(‘c, expr, expr3) = p(‘c, expression, conditional)`
- `p(‘c, expr1, seq([expr, ops, expr])) = p(‘c, binary, seq([expression, operator, expression]))`
- `p(‘c, expr2, seq([ID, +(expr)])) = p(‘c, apply, seq([str, +(expression)]))`
- `p(‘c, expr3, seq([expr, expr, expr])) = p(‘c, conditional, seq([expression, expression, expression]))`

This yields the following nominal mapping:

\[ \text{antlr} \circ \text{master} = \{ \langle \text{program, program}, \rangle, \langle \text{expr3, conditional}, \rangle, \langle \text{expr1, binary}, \rangle, \langle \text{function, function}, \rangle, \langle \text{ID, str}, \rangle, \langle \text{expr, expression}, \rangle, \langle \text{INT, int}, \rangle, \langle \text{ops, operator}, \rangle, \langle \text{NEWLINE, } \omega, \rangle, \langle \text{expr2, apply} \rangle \} \]

Which is exercised with these grammar transformation steps:

- `renameN-renameN expr3 to conditional`
- `renameN-renameN expr1 to binary`
- `renameN-renameN ID to str`
- `renameN-renameN expr to expression`
- `renameN-renameN INT to int`
- `renameN-renameN ops to operator`
- `renameN-renameN expr2 to apply`
1.6 Structural resolution

- project-inject

\[
p('',function,seq([str, +(str), expression, +(NEWLINE)]))\
\]
Grammar 2

Definite Clause Grammar

Source name: dcg

2.1 Source grammar

- Source artifact: topics/fl/prolog1/Parser.pro
- Grammar extractor: shared/prolog/cli/dcg2bgf.pro

| Production rules |
|------------------|
| p('', program, +(function)) |
| p('', function, seq([name, +(name), '(', expr, +(newline)]))) |
| p('binary', expr, seq([atom, +(seq([ops, atom]))])) |
| p('ifThenElse', expr, seq([('if', expr, 'then', expr, 'else', expr)])) |
| p('literal', atom, int) |
| p('argument', atom, name) |
| p('=', atom, seq([('(', expr, ')')])) |
| p('equal', ops, '==') |
| p('plus', ops, '+') |
| p('minus', ops, '-') |

2.2 Mutations

- *unite-splitN* expr
  - p('literal', atom, int)
  - p('argument', atom, name)
  - p('=', atom, seq([('(', expr, ')')]))
- *assoc-iterate*
  - p('binary', expr, seq([expr, ops, expr]))

2.3 Normalizations

- *reroot-reroot* [] to [program]
- *unlabel-designate*
  - p('binary', expr, seq([expr, ops, expr]))
- *unlabel-designate*
  - p('apply', expr, seq([name, +(expr)]))
- *unlabel-designate*
  - p('ifThenElse', expr, seq([('if', expr, 'then', expr, 'else', expr)]))
- *unlabel-designate*
  - p('literal', expr, int)
• unlabeled-designate
  p ('argument', expr, name)

• unlabeled-designate
  p ('equal', ops, '==')

• unlabeled-designate
  p ('plus', ops, '+')

• unlabeled-designate
  p ('minus', ops, '-')

• abstractize-concretize
  p ('', expr, seq ([('(', expr, ')')]))

• abstractize-concretize
  p ('', function, seq ([name, +(name), ('==', expr, +(newline))]))

• abstractize-concretize
  p ('', ops, '==')

• abstractize-concretize
  p ('', expr, seq ([('if', expr, 'then', expr, 'else', expr)]))

• abstractize-concretize
  p ('', ops, '+')

• abstractize-concretize
  p ('', ops, '-')

• abstractize-concretize
  p ('', expr, seq ([name, +(expr)]))

• abstractize-concretize
  p ('', expr, seq ([expr, +(expr), expr]))

• extract-inline in expr
  p ('', expr, seq ([expr, ops, +(expr), expr]))

• extract-inline in expr
  p ('', expr, seq ([name, +(expr)]))

• extract-inline in expr
  p ('', expr, seq ([expr, expr, expr]))

  2.4 Grammar in ANF

| Production rule | Production signature |
|-----------------|----------------------|
| p('', program, +(function)) | {⟨function, +⟩} |
| p('', function, seq ([name, +(name), expr, +(newline)])) | {⟨expr, 1⟩, ⟨newline, +⟩, ⟨name, 1+⟩} |
| p('', expr, expr1) | {⟨expr1, 1⟩} |
| p('', expr, expr2) | {⟨expr2, 1⟩} |
| p('', expr, expr3) | {⟨expr3, 1⟩} |
| p('', expr, int) | {⟨int, 1⟩} |
| p('', expr, name) | {⟨name, 1⟩} |
| p('', expr1, seq ([expr, ops, expr2])) | {⟨ops, 1⟩, ⟨expr, 11⟩} |
| p('', expr2, seq ([name, +(expr)])) | {⟨expr, +⟩, ⟨name, 1⟩} |
| p('', expr3, seq ([expr, expr, expr])) | {⟨expr, 111⟩} |
2.5 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

\[
\begin{align*}
p('', \text{program}, +\text{(function)}) & \simeq p('', \text{program}, +\text{(function)}) \\
p('', \text{function}, \text{seq}([[\text{name}, +(\text{name})], \text{expr}, -(\text{newline})]]) & \simeq p('', \text{function}, \text{seq}([\text{str}, +(\text{str}), \text{expression} })) \\
p('', \text{expr}, \text{expr}_1) & \simeq p('', \text{expression}, \text{binary}) \\
p('', \text{expr}, \text{expr}_2) & \simeq ('', \text{expression}, \text{apply}) \\
p('', \text{expr}, \text{expr}_3) & \simeq ('', \text{expression}, \text{conditional}) \\
p('', \text{expr}, \text{int}) & \simeq ('', \text{expression}, \text{int}) \\
p('', \text{expr}, \text{name}) & \simeq ('', \text{expression}, \text{str}) \\
p('', \text{expr}_1, \text{seq}([[\text{expr}, \text{ops}], \text{expr}])) & \simeq ('', \text{binary}, \text{seq}([\text{expression}, \text{operator}, \text{expression}])) \\
p('', \text{expr}_2, \text{seq}([\text{name}, -(\text{expr})])) & \simeq ('', \text{apply}, \text{seq}([\text{str}, -(\text{expression})])) \\
p('', \text{expr}_3, \text{seq}([\text{expr}, \text{expr}], \text{expr})) & \simeq ('', \text{conditional}, \text{seq}([\text{expression}, \text{expression}, \text{expression}]))
\end{align*}
\]

This yields the following nominal mapping:

\[
dcg \odot \text{master} = \\
\{(\text{program, program}), \\
(\text{expr}_3, \text{conditional}), \\
(\text{expr}_1, \text{binary}), \\
(\text{function, function}), \\
(\text{expr}, \text{expression}), \\
(\text{name, str}), \\
(\text{ops, operator}), \\
(\text{int, int}), \\
(\text{newline, }\omega), \\
(\text{expr}_2, \text{apply})\}
\]

Which is exercised with these grammar transformation steps:

- rename\(N\)-rename\(N\) \text{expr}_3 to \text{conditional}
- rename\(N\)-rename\(N\) \text{expr}_1 to \text{binary}
- rename\(N\)-rename\(N\) \text{expr} to \text{expression}
- rename\(N\)-rename\(N\) \text{name} to \text{str}
- rename\(N\)-rename\(N\) \text{ops} to \text{operator}
- rename\(N\)-rename\(N\) \text{int} to \text{int}
- rename\(N\)-rename\(N\) \text{expr}_2 to \text{apply}

2.6 Structural resolution

- project-inject
  \[
p('', \text{function}, \text{seq}([\text{str}, +(\text{str}), \text{expression}, +(\text{newline})]))
  \]
Grammar 3

Eclipse Modeling Framework

Source name: emf

3.1 Source grammar

- Source artifact: topics/fl/emf2/model/fl.ecore
- Grammar extractor: topics/extraction/ecore/ecore2bgf.xslt

| Production rules |
|------------------|
| p("", Apply, seq([sel ("name", str), +(sel ("arg", Expr))])) |
| p("", Argument, sel ("name", str)) |
| p("", Binary, seq([sel ("ops", Ops), sel ("left", Expr), sel ("right", Expr)])) |
| p("", Expr, choice([Apply, Argument, Binary, IfThenElse, Literal])) |
| p("", Function, seq([sel ("name", str), +(sel ("arg", str)), sel ("rhs", Expr)])) |
| p("", IfThenElse, seq([sel ("ifExpr", Expr), sel ("thenExpr", Expr), sel ("elseExpr", Expr)])) |
| p("", Literal, sel ("info", int)) |
| p("", Ops, choice([sel ("Equal", ε), sel ("Plus", ε), sel ("Minus", ε)])) |
| p("", ProgramType, +(sel ("function", Function))) |

3.2 Normalizations

- reroot-reroot [] to [ProgramType]
- unlabel-designate
  p ("name", Argument, str)
- unlabel-designate
  p ("info", Literal, int)
- anonymize-deanonymize
  p ("", Function, seq([sel ("name", str), +(sel ("arg", str)), sel ("rhs", Expr)]))
- anonymize-deanonymize
  p ("", Apply, seq([sel ("name", str), +(sel ("arg", Expr)]))
- anonymize-deanonymize
  p ("", IfThenElse, seq([sel ("ifExpr", Expr), sel ("thenExpr", Expr), sel ("elseExpr", Expr)]))
• anonymize-deanonymize
  \[ p (\text{""}, \text{Ops}, \text{choice (sel ("Equal", ε) sel ("Plus", ε) sel ("Minus", ε)})) \]

• anonymize-deanonymize
  \[ p (\text{""}, \text{ProgramType}, +\{(\text{sel ("Function", \text{Function})})\}) \]

• anonymize-deanonymize
  \[ p (\text{""}, \text{Binary}, \text{seq (sel ("ops", \text{Ops}) sel ("left", \text{Expr}) sel ("right", \text{Expr})})}) \]

• vertical-horizontal in \text{Expr}
• undefined-define
  \[ p (\text{""}, \text{Ops}, ε) \]

• unchain-chain
  \[ p (\text{""}, \text{Expr, Apply}) \]

• unchain-chain
  \[ p (\text{""}, \text{Expr, Argument}) \]

• unchain-chain
  \[ p (\text{""}, \text{Expr, Binary}) \]

• unchain-chain
  \[ p (\text{""}, \text{Expr, IfThenElse}) \]

• unchain-chain
  \[ p (\text{""}, \text{Expr, Literal}) \]

• unlabel-designate
  \[ p (\text{"Apply"}, \text{Expr}, \text{seq ([str, +\{(\text{Expr})}\})}) \]

• unlabel-designate
  \[ p (\text{"Argument"}, \text{Expr}, \text{str}) \]

• unlabel-designate
  \[ p (\text{"Binary"}, \text{Expr, seq ([\text{Ops, Expr, Expr}])}) \]

• unlabel-designate
  \[ p (\text{"IfThenElse"}, \text{Expr, seq ([Expr, Expr, Expr])}) \]

• unlabel-designate
  \[ p (\text{"Literal"}, \text{Expr, int}) \]

• extract-inline in \text{Expr}
  \[ p (\text{""}, \text{Expr, seq ([str, +\{(\text{Expr})\})}) \]

• extract-inline in \text{Expr}
  \[ p (\text{""}, \text{Expr2, seq ([\text{Ops, Expr, Expr}])}) \]

• extract-inline in \text{Expr}
  \[ p (\text{""}, \text{Expr3, seq ([Expr, Expr, Expr])}) \]

3.3 Grammar in ANF

| Production rule | Production signature |
|-----------------|----------------------|
| \[ p (\text{""}, \text{Expr, Expr1}) \] | \{ (\text{Expr1, 1}) \} |
| \[ p (\text{""}, \text{Expr, str}) \] | \{ (\text{str, 1}) \} |
| \[ p (\text{""}, \text{Expr, Expr2}) \] | \{ (\text{Expr2, 1}) \} |
| \[ p (\text{""}, \text{Expr, Expr3}) \] | \{ (\text{Expr3, 1}) \} |
| \[ p (\text{""}, \text{Expr, int}) \] | \{ (\text{int, 1}) \} |
| \[ p (\text{""}, \text{Function, seq ([str, +\{\text{str, Expr}\}])}) \] | \{ (str, 1+), (\text{Expr, 1}) \} |
| \[ p (\text{""}, \text{ProgramType, +\{Function\})} \] | \{ (Function, +) \} |
| \[ p (\text{""}, \text{Expr1, seq ([\text{Expr}, +\{\text{Expr}\}])}) \] | \{ (\text{str, 1}), (\text{Expr, +}) \} |
| \[ p (\text{""}, \text{Expr2, seq ([Ops, Expr, Expr])}) \] | \{ (Ops, 1), (Expr, 11) \} |
| \[ p (\text{""}, \text{Expr3, seq ([Expr, Expr, Expr])}) \] | \{ (Expr, 111) \} |
3.4 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

\[
\begin{align*}
p('', \text{Expr}, \text{Expr}_1) & \equiv p('', \text{expression}, \text{apply}) \\
p('', \text{Expr}, \text{str}) & \equiv p('', \text{expression}, \text{str}) \\
p('', \text{Expr}, \text{Expr}_2) & \equiv p('', \text{expression}, \text{binary}) \\
p('', \text{Expr}, \text{Expr}_3) & \equiv p('', \text{expression}, \text{conditional}) \\
p('', \text{Expr}, \text{int}) & \equiv p('', \text{expression}, \text{int}) \\
p('', \text{Function}, \text{seq}([\text{str}, +, \text{Expr}])) & \equiv p('', \text{function}, \text{seq}([\text{str}, +, \text{expression}])) \\
p('', \text{ProgramType}, +, \text{Function}) & \equiv p('', \text{program}, +, \text{function}) \\
p('', \text{Expr}_1, \text{seq}([\text{str}, +(\text{Expr})])) & \equiv p('', \text{apply}, \text{seq}([\text{str}, +(\text{expression})))) \\
p('', \text{Expr}_2, \text{seq}([\text{Ops}, \text{Expr}, \text{Expr}])) & \equiv p('', \text{binary}, \text{seq}([\text{expression}, \text{operator}, \text{expression}])) \\
p('', \text{Expr}_3, \text{seq}([\text{Expr}, \text{Expr}, \text{Expr}])) & \equiv p('', \text{conditional}, \text{seq}([\text{expression}, \text{expression}, \text{expression}]))
\end{align*}
\]

This yields the following nominal mapping:

\[
\begin{align*}
\text{emf} \odot \text{master} = \{ & \langle \text{Expr}_2, \text{binary} \rangle, \\
& \langle \text{ProgramType}, \text{program} \rangle, \\
& \langle \text{Expr}_3, \text{conditional} \rangle, \\
& \langle \text{str}, \text{str} \rangle, \\
& \langle \text{int}, \text{int} \rangle, \\
& \langle \text{Function}, \text{function} \rangle, \\
& \langle \text{Expr}, \text{expression} \rangle, \\
& \langle \text{Expr}_1, \text{apply} \rangle, \\
& \langle \text{Ops}, \text{operator} \rangle \}
\end{align*}
\]

Which is exercised with these grammar transformation steps:

- renameN-renameN Expr2 to binary
- renameN-renameN ProgramType to program
- renameN-renameN Expr3 to conditional
- renameN-renameN Function to function
- renameN-renameN Expr to expression
- renameN-renameN Expr1 to apply
- renameN-renameN Ops to operator

3.5 Structural resolution

- permute-permute

\[
\begin{align*}
p('', \text{binary}, \text{seq}([\text{operator}, \text{expression}, \text{expression}])) & \\
p('', \text{binary}, \text{seq}([\text{expression}, \text{operator}, \text{expression}]))
\end{align*}
\]
Grammar 4

JAXB Data Binding Framework

Source name: jaxb

4.1 Source grammar

- Source artifact: topics/fl/java3/fl/Apply.java
- Source artifact: topics/fl/java3/fl/Argument.java
- Source artifact: topics/fl/java3/fl/Binary.java
- Source artifact: topics/fl/java3/fl/Expr.java
- Source artifact: topics/fl/java3/fl/Function.java
- Source artifact: topics/fl/java3/fl/IfThenElse.java
- Source artifact: topics/fl/java3/fl/Literal.java
- Source artifact: topics/fl/java3/fl/ObjectFactory.java
- Source artifact: topics/fl/java3/fl/Ops.java
- Source artifact: topics/fl/java3/fl/Program.java
- Source artifact: topics/fl/java3/fl/package-info.java
- Grammar extractor: topics/extraction/java2bgf/slps/java2bgf/Tool.java

| Production rules |
|------------------|
| p(′), Apply, seq (sel (′Name′, str), sel (′Arg′, *(Expr)))) |
| p(′), Argument, sel (′Name′, str)) |
| p(′), Binary, seq (sel (′Ops′, Ops), sel (′Left′, Expr), sel (′Right′, Expr))) |
| p(′), Expr, choice([Apply, Argument, Binary, IfThenElse, LITERAL]) |
| p(′), Function, seq (sel (′Name′, str), sel (′Arg′, *(str)), sel (′Rhs′, Expr))) |
| p(′), IfThenElse, seq (sel (′IfExpr′, Expr), sel (′ThenExpr′, Expr), sel (′ElseExpr′, Expr))) |
| p(′), Literal, sel (′Info′, int)) |
| p(′), ObjectFactory, e) |
| p(′), Ops, choice([sel (′EQUAL′, e), sel (′PLUS′, e), sel (′MINUS′, e)]) |
| p(′), package − info, ϕ) |
| p(′), Program, sel (′Function′, *(Function))) |
4.2 Normalizations

- reroot-reroot [] to [Program]
- unlabel-designate
  p ([Name] Argument, str)
- unlabel-designate
  p ([Info] Literal, int)
- unlabel-designate
  p ([Function] Program, *(Function))
- anonymize-deanonymize
  p ("", IfThenElse, seq ([sel("IfExpr", Expr) sel("ThenExpr", Expr) sel("ElseExpr", Expr)]))
- anonymize-deanonymize
  p ("", Function, seq ([sel("Name", str) sel("Arg", *(str)) sel("Rhs", Expr)]))
- anonymize-deanonymize
  p ("", Binary, seq ([sel("Ops", Ops) sel("Left", Expr) sel("Right", Expr)]))
- anonymize-deanonymize
  p ("", Apply, seq ([sel("Name", str) sel("Arg", *(Expr)]))
- anonymize-deanonymize
  p ("", Ops, choice ([sel("EQUAL", ε) sel("PLUS", ε) sel("MINUS", ε)]))
- vertical-horizontal in Expr
- undefined-define
  p ("", Ops, ε)
- eliminate-introduce
  p ("", ObjectFactory, ε)
- eliminate-introduce
  p ("", package − info, φ)
- unchain-chain
  p ("", Expr, Apply)
- unchain-chain
  p ("", Expr, Argument)
- unchain-chain
  p ("", Expr, Binary)
- unchain-chain
  p ("", Expr, IfThenElse)
- unchain-chain
  p ("", Expr, Literal)
- unlabel-designate
  p ([Apply] Expr, seq ([str, *(Expr)]))
- unlabel-designate
  p ([Argument] Expr, str)
- unlabel-designate
  p ([Binary] Expr, seq ([Ops, Expr, Expr]))
- unlabel-designate
  p ([IfThenElse] Expr, seq ([Expr, Expr, Expr]))
- unlabel-designate
  p ([Literal] Expr, int)
- extract-inline in Expr
  p ("", Expr, seq ([str, *(Expr)]))
- extract-inline in Expr
  p ("", Expr, seq ([Ops, Expr, Expr]))
- extract-inline in Expr
  p ("", Expr, seq ([Expr, Expr, Expr]))
4.3 Grammar in ANF

| Production rule          | Production signature |
|--------------------------|----------------------|
| \( p('ε', Expr, Expr1)  \) | \{\{Expr1, 1\}\}    |
| \( p('ε', Expr, str)    \)  | \{\{str, 1\}\}     |
| \( p('ε', Expr, Expr2)  \) | \{\{Expr2, 1\}\}    |
| \( p('ε', Expr, Expr3)  \) | \{\{Expr3, 1\}\}    |
| \( p('ε', Expr, int)    \)  | \{\{int, 1\}\}     |
| \( p('ε', Function, seq ([str, *(str), Expr])) \) | \{\{Expr1, 1, \{str, 1\}\}\} |
| \( p('ε', Program, *(Function)) \) | \{\{Function, \}\} |
| \( p('ε', Expr1, seq ([str, *(Expr)])) \) | \{\{str, 1, \{Expr, \}\}\} |
| \( p('ε', Expr2, seq ([Ops, Expr, Expr1])) \) | \{\{Ops, 1, \{Expr, 1\}\}\} |
| \( p('ε', Expr3, seq ([Expr, Expr, Expr1])) \) | \{\{Expr, 111\}\} |

4.4 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

\[
\begin{align*}
p('ε', Expr, Expr1) & \Rightarrow p('ε', \text{expression}, \text{apply}) \\
p('ε', Expr, str) & \Rightarrow p('ε', \text{expression}, \text{str}) \\
p('ε', Expr, Expr2) & \Rightarrow p('ε', \text{expression}, \text{binary}) \\
p('ε', Expr, Expr3) & \Rightarrow p('ε', \text{expression}, \text{conditional}) \\
p('ε', Expr, int) & \Rightarrow p('ε', \text{expression}, \text{int}) \\
p('ε', Function, seq ([str, *(str), Expr])) & \Rightarrow p('ε', \text{function}, seq ([str, *(str), \text{expression}])) \\
p('ε', Program, *(Function)) & \Rightarrow p('ε', \text{program}, *(Function)) \\
p('ε', Expr1, seq ([str, *(Expr)])) & \Rightarrow p('ε', \text{apply}, seq ([str, *(expression)])) \\
p('ε', Expr2, seq ([Ops, Expr, Expr1])) & \Rightarrow p('ε', \text{binary}, seq ([\text{expression}, \text{operator}, \text{expression}])) \\
p('ε', Expr3, seq ([Expr, Expr, Expr1])) & \Rightarrow p('ε', \text{conditional}, seq ([\text{expression}, \text{expression}, \text{expression}]))
\end{align*}
\]

This yields the following nominal mapping:

\[
jab \circ \text{master} = \{(\text{Expr2, binary}), \ (\text{Expr3, conditional}), \ (\text{int, int}), \ (\text{Function, function}), \ (\text{str, str}), \ (\text{Program, program}), \ (\text{Expr, expression}), \ (\text{Expr1, apply}), \ (\text{Ops, operator})\}
\]

Which is exercised with these grammar transformation steps:

- renameN-renameN Expr2 to binary
- renameN-renameN Expr3 to conditional
- renameN-renameN Function to function
- renameN-renameN Program to program
- renameN-renameN Expr to expression
- renameN-renameN Expr1 to apply
- renameN-renameN Ops to operator

4.5 Structural resolution

- narrow-widen in function
  \( \text{*(str)} \)
  \( \text{~*(str)} \)
• **narrow-widen** in *program*
  \*\(\text{function}\)  
  \+\(\text{function}\)

• **narrow-widen** in *apply*
  \*\(\text{expression}\)  
  \+\(\text{expression}\)

• **permute-permute**
  \p{\('', \text{binary}, \text{seq}([\text{operator}, \text{expression}, \text{expression}])\)}
  \p{\('', \text{binary}, \text{seq}([\text{expression}, \text{operator}, \text{expression}])\)}
Grammar 5

Java Object Model

Source name: om

5.1 Source grammar

- Source artifact: topics/fl/java1/types/Apply.java
- Source artifact: topics/fl/java1/types/Argument.java
- Source artifact: topics/fl/java1/types/Binary.java
- Source artifact: topics/fl/java1/types/Expr.java
- Source artifact: topics/fl/java1/types/Function.java
- Source artifact: topics/fl/java1/types/IfThenElse.java
- Source artifact: topics/fl/java1/types/Literal.java
- Source artifact: topics/fl/java1/types/Ops.java
- Source artifact: topics/fl/java1/types/Program.java
- Source artifact: topics/fl/java1/types/Visitor.java
- Grammar extractor: topics/extraction/java2bgf/slps/java2bgf/Tool.java

| Production rules |
|------------------|
| `p(,, Apply, seq ([sel (‘name’, str), sel (‘args’, *(Expr))])))` |
| `p(,, Argument, sel (‘name’, str))` |
| `p(,, Binary, seq ([sel (‘ops’, Ops), sel (‘left’, Expr), sel (‘right’, Expr))])` |
| `p(,, Expr, choice ([Apply, Argument, Binary, IfThenElse, Literal]))` |
| `p(,, Function, seq ([sel (‘name’, str), sel (‘args’, *(str)), sel (‘rhs’, Expr))])` |
| `p(,, IfThenElse, seq ([sel (‘ifExpr’, Expr), sel (‘thenExpr’, Expr), sel (‘elseExpr’, Expr))])` |
| `p(,, Literal, sel (‘info’, int))` |
| `p(,, Ops, choice ([sel (‘Equal’, ε), sel (‘Plus’, ε), sel (‘Minus’, ε))])` |
| `p(,, Program, sel (‘functions’, *(Function)))]` |
| `p(,, Visitor, ϕ)` |

5.2 Normalizations

- reroot-reroot [] to [Program]
- unlabel-designate
  - p [name] Argument, str
• unlabel-designate
  p ('info', Literal, int)

• unlabel-designate
  p ('functions', Program, *(Function))

• anonymize-deanonymize
  p ('', IfThenElse, seq([sel('ifExpr', Expr) sel('thenExpr', Expr) sel('elseExpr', Expr)]))

• anonymize-deanonymize
  p ('', Ops, choice([sel('Equal', ε) sel('Plus', ε) sel('Minus', ε)]))

• anonymize-deanonymize
  p ('', Function, seq([sel('name', str) sel('args', *(Expr)) sel('rhs', Expr)]))

• anonymize-deanonymize
  p ('', Binary, seq([sel('ops', Ops) sel('left', Expr) sel('right', Expr)]))

• vertical-horizontal in Expr

• eliminate-introduce
  p ('', Visitor, ϕ)

• undefine-define
  p ('', Ops, ε)

• unchain-chain
  p ('', Expr, Apply)

• unchain-chain
  p ('', Expr, Argument)

• unchain-chain
  p ('', Expr, Binary)

• unchain-chain
  p ('', Expr, IfThenElse)

• unchain-chain
  p ('', Expr, Literal)

• unlabel-designate
  p ('Apply', Expr, seq([str, *(Expr)]))

• unlabel-designate
  p ('Argument', Expr, str)

• unlabel-designate
  p ('Binary', Expr, seq([Ops, Expr, Expr]))

• unlabel-designate
  p ('IfThenElse', Expr, seq([Expr, Expr, Expr]))

• unlabel-designate
  p ('Literal', Expr, int)

• extract-inline in Expr
  p ('', Expr, seq([str, *(Expr)]))

• extract-inline in Expr
  p ('', Expr, seq([Ops, Expr, Expr]))

• extract-inline in Expr
  p ('', Expr, seq([Expr, Expr, Expr]))
5.3 Grammar in ANF

| Production rule | Production signature |
|-----------------|----------------------|
| \(p(\text{Expr, Expr})\) | \{\{Expr, Expr\}\} |
| \(p(\text{Expr, str})\) | \{\{str\}\} |
| \(p(\text{Expr, Expr2})\) | \{\{Expr2\}\} |
| \(p(\text{Expr, Expr3})\) | \{\{Expr, Expr\}\} |
| \(p(\text{Expr, int})\) | \{\{int\}\} |
| \(p(\text{Function, seq ([str, +(str), Expr]])\) | \{\{Expr1, \{(str), Expr\}\}\} |
| \(p(\text{Program, +(Function)})\) | \{\{Function\}\} |
| \(p(\text{Expr1, seq ([str, +(Expr)])})\) | \{\{str, \{Expr\}\}\} |
| \(p(\text{Expr2, seq ([Ops, Expr, Expr])})\) | \{\{Ops, \{Expr\}\}\} |
| \(p(\text{Expr3, seq ([Expr, Expr, Expr])})\) | \{\{Expr, \{Expr, Expr\}\}\} |

This yields the following nominal mapping:

\[\text{nom} \circ \text{master} = \{(\text{Expr2, binary}), \]  
\{(Expr3, conditional), \}  
\{(int, int), \}  
\{(Function, function), \}  
\{(str, str), \}  
\{(Program, program), \}  
\{(Expr, expression), \}  
\{(Expr1, apply), \}  
\{(Ops, operator)\}\]

Which is exercised with these grammar transformation steps:

- renameN-renameN Expr2 to binary
- renameN-renameN Expr3 to conditional
- renameN-renameN Function to function
- renameN-renameN Program to program
- renameN-renameN Expr to expression
- renameN-renameN Expr1 to apply
- renameN-renameN Ops to operator

5.4 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

\[\begin{align*}
\text{p (Expr, Expr1)} &\quad=\quad \text{p ('\text{expression, apply}')}
\text{p (Expr, str)} &\quad=\quad \text{p ('\text{expression, str}')}
\text{p (Expr, Expr2)} &\quad=\quad \text{p ('\text{expression, binary}')}
\text{p (Expr, Expr3)} &\quad=\quad \text{p ('\text{expression, conditional}')}
\text{p (Expr, int)} &\quad=\quad \text{p ('\text{expression, int}')}
\text{p (Function, seq ([str, +(str), Expr]))} &\quad=\quad \text{p ('\text{function, seq ([str, +(str), expression'])}}
\text{p (Program, +(Function))} &\quad=\quad \text{p ('\text{program, +(function)}')}
\text{p (Expr1, seq ([str, +(Expr)])}) &\quad=\quad \text{p ('\text{apply, seq ([str, +(expression)])}')}
\text{p (Expr2, seq ([Ops, Expr, Expr])}) &\quad=\quad \text{p ('\text{binary, seq ([expression, operator, expression'])}}
\text{p (Expr3, seq ([Expr, Expr, Expr])}) &\quad=\quad \text{p ('\text{conditional, seq ([expression, expression, expression'])}}
\end{align*}\]

5.5 Structural resolution

- narrow-widen in function
  \(+(\text{str})\)
  \(-\text{str}\)
• **narrow-widen** in `program`
  `*(function)`
  `=|(function)`

• **narrow-widen** in `apply`
  `*(expression)`
  `=|(expression)`

• **permute-permute**
  `p('', binary, seq([operator, expression, expression])))`
  `p('', binary, seq([expression, operator, expression])))`
Grammar 6

PyParsing in Python

Source name: python

6.1 Source grammar

- Source artifact: topics/B/python/parser.py
- Grammar extractor: shared/rascal/src/extract/Python2BGF.rsc

| Production rules |
|------------------|
| `p(\', Literal, Literal)` |
| `p(\', IF, 'if')` |
| `p(\', THEN, 'then')` |
| `p(\', ELSE, 'else')` |
| `p(\', name, str)` |
| `p(\', literal, seq ([?('-'), int]))` |
| `p(\', atom, choice([name, literal, seq ([?('-'), int])]))` |
| `p(\', ifThenElse, seq ([IF, expr, THEN, expr, ELSE, expr]))` |
| `p(\', operators, choice([==, +, -]))` |
| `p(\', binary, seq ([atom, *seq ([operators, atom])]))` |
| `p(\', apply, seq ([name, *seq [atom]]))` |
| `p(\', expr, choice([binary, apply, ifThenElse]))` |
| `p(\', function, seq ([name, *(name), '=', expr]))` |
| `p(\', program, seq ([:(function), StringEnd]))` |

6.2 Mutations

- `unite-splitN expr`
  `p(\', atom, choice ([name, literal, seq ([?('-'), int])]))`

- `designate-unlabel`
  `p('tmplabel', binary, seq ([expr, *seq ([operators, expr])]))`

- `assoc-iterate`
  `p('tmplabel', binary, seq ([expr, operators, expr]))`

- `unlabel-designate`
  `p('tmplabel', binary, seq ([expr, operators, expr]))`
6.3 Normalizations

- reroot-reroot [] to [program]
- abstractize-concretize
  p (', literal, seq ([? [ ] ; int]))
- abstractize-concretize
  p (', operators, choice ([ == [ ] ] + [ ] ))
- abstractize-concretize
  p (', IF, [ ] )
- abstractize-concretize
  p (', expr, choice ([ name, literal, seq ([ ( [ expr ] ] ) ] ) ] )
- abstractize-concretize
  p (', ELSE, [ ] )
- abstractize-concretize
  p (', THEN, [ ] )
- abstractize-concretize
  p (', function, seq ([ name, +(name), [ == [ ] ] expr ] ) )
- vertical-horizontal in expr
- undefine-define
  p (', IF, ε)
- undefine-define
  p (', THEN, ε)
- undefine-define
  p (', ELSE, ε)
- undefine-define
  p (', operators, ε)
- unchain-chain
  p (', expr, literal)
- abridge-detour
  p (', expr, expr)
- unchain-chain
  p (', expr, binary)
- unchain-chain
  p (', expr, apply)
- unchain-chain
  p (', expr, ifThenElse)
- inline-extract
  p (', name, str)
- unlabel-designate
  p (', literal expr, int)
- unlabel-designate
  p (', IFThenElse expr, seq ([ IF, expr, THEN, expr, ELSE, expr ] ) )
- unlabel-designate
  p (', binary expr, seq ([ expr, operators, expr ] ) )
- unlabel-designate
  p (', apply expr, seq ([ str, +(expr) ] ) )
- extract-inline in expr
  p (', expr1, seq ([ IF, expr, THEN, expr, ELSE, expr ] ) )
- extract-inline in expr
  p (', expr2, seq ([ expr, operators, expr ] ) )
- extract-inline in expr
  p (', expr3, seq ([ str, +(expr) ] ) )
6.4 Grammar in ANF

| Production rule | Production signature |
|-----------------|----------------------|
| `p (', Literal, Literal)` | `{(Literal, 1)}` |
| `p (', expr, int)` | `{(int, 1)}` |
| `p (', expr, str)` | `{(str, 1)}` |
| `p (', expr, expr)` | `{(expr, 1)}` |
| `p (', expr, expr)` | `{(expr, 2)}` |
| `p (', expr, expr)` | `{(expr, 3)}` |
| `p (', function, seq {[str, +(str), expr]})` | `{(str, 1+), (expr, 1)}` |
| `p (', expr, seq {[IF, expr, THEN, expr, ELSE, expr]})` | `{(IF, 1), (THEN, 1), (expr, 111), (ELSE, 1)}` |
| `p (', expr, seq {[expr, operators, expr]})` | `{(expr, 11), (operators, 1)}` |
| `p (', expr, seq {[str, -(expr)]})` | `{(str, 1), (expr, +)}` |

6.5 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

```
p (', , Literal, Literal) ≅
p (', expr, int) = p ('', expression, int)
p (', expr, str) = p ('', expression, str)
p (', expr, expr) = p ('', expression, conditional)
p (', expr, expr) = p ('', expression, binary)
p (', expr, expr) = p ('', expression, apply)
p (', function, seq {[str, +(str), expression]}) = p ('', function, seq {[str, +(str), expression]})
p (', function, seq {[str, +(str), expression]}) = p ('', program, +{function})
p (', expr, seq {[IF, expr, THEN, expr, ELSE, expr]}) = p ('', conditional, seq {[expression, expression, expression]})
p (', expr, seq {[expr, operators, expr]}) = p ('', binary, seq {[expression, operator, expression]})
p (', expr, seq {[str, +(expr)]}) = p ('', apply, seq {[str, +(expression)]})
```

This yields the following nominal mapping:

```
python ◦ master = { (expr_2, binary),
(program, program),
(function, function),
(expr_1, conditional),
(expr, expression),
(str, str),
(int, int),
(StringEnd, ω),
(ELSE, ω),
(IF, ω),
(expr, apply),
(THEN, ω),
(operators, operator)
```

Which is exercised with these grammar transformation steps:

- `renameN-renameN expr_2 to binary`
- `renameN-renameN expr_1 to conditional`
- `renameN-renameN expr to expression`
- `renameN-renameN expr to apply`
- `renameN-renameN operators to operator`
6.6 Structural resolution

- project-inject
  $p('', \text{program}, \text{seq}\{\lnot \text{function}, \text{StringEnd}\})$

- project-inject
  $p('', \text{conditional}, \text{seq}\{\text{IF}, \text{expression}, \text{THEN}, \text{expression}, \text{ELSE}, \text{expression}\})$

- project-inject
  $p('', \text{conditional}, \text{seq}\{\text{IF}, \text{expression}, \text{THEN}, \text{expression}, \text{ELSE}, \text{expression}\})$

- project-inject
  $p('', \text{conditional}, \text{seq}\{\text{expression, THEN, expression, expression}\})$

- eliminate-introduce
  $p('', \text{Literal, Literal})$
Grammar 7

Rascal Algebraic Data Type

Source name: rascal-a

7.1 Source grammar

- Source artifact: topics/fl/rascal/Abstract.rsc
- Grammar extractor: shared/rascal/src/extract/RascalADT2BGF.rsc

```
| Production rules |
|------------------|
| p('prg', FLPrg, sel('fs', *(FLFun))) |
| p('fun', FLPrg, seq (sel('f', str), sel('args', *(str)) ), sel('body', FLEexpr))) |
| p('fun', FLEexpr, choice([sel('binary', seq ([sel('e1', FLEexpr), sel('op', FLOp), sel('e2', FLEexpr)]), sel('apply', seq ([sel('f', str), sel('vargs', *(FLexpr))]), sel('ifThenElse', seq ([sel('c', FLEexpr), sel('t', FLEexpr), sel('e', FLEexpr)]), sel('argument', sel('a', str)), sel('literal', sel('i', int)))])) |
| p('op', FLOp, choice([sel('minus', ε), sel('plus', ε), sel('equal', ε)])) |
| p('expr', FLexpr, choice([sel('binary', seq ([sel('e1', FLEexpr), sel('op', FLOp), sel('e2', FLEexpr)]), sel('apply', seq ([sel('f', str), sel('vargs', *(FLexpr))]), sel('ifThenElse', seq ([sel('c', FLEexpr), sel('t', FLEexpr), sel('e', FLEexpr)]), sel('argument', sel('a', str)), sel('literal', sel('i', int)))])) |
```

7.2 Normalizations

- reroot-reroot [] to [FLPrg]
- unlabel-designate
  ```
p('prg', FLPrg, sel('fs', *(FLFun)))
```
- unlabel-designate
  ```
p('fun', FLPrg, seq (sel('f', str), sel('args', *(str)) ), sel('body', FLEexpr)))
```
- anonymize-deanonymize
  ```
p('op', FLOp, choice([sel('minus', ε), sel('plus', ε), sel('equal', ε)]))
```
- anonymize-deanonymize
  ```
p('expr', FLEexpr, choice([sel('binary', seq ([sel('e1', FLEexpr), sel('op', FLOp), sel('e2', FLEexpr)]), sel('apply', seq ([sel('f', str), sel('vargs', *(FLexpr))]), sel('ifThenElse', seq ([sel('c', FLEexpr), sel('t', FLEexpr), sel('e', FLEexpr)]), sel('argument', sel('a', str)), sel('literal', sel('i', int)))]))
```
- anonymize-deanonymize
  ```
p('fun', seq ([sel('t', str), sel('args', *(str)) ], sel('body', FLEexpr)]))
```
- vertical-horizontal in FLEexpr
- undefined-define
  ```
p('expr', FLOp, ε)
```
- unlabel-designate
  ```
p('prg', FLPrg, *(FLFun))
```
This yields the following nominal mapping:

\[p \ ('' \ , \ FLFun \ ) \ \mapsto \ \{ \langle FLFun, \ast \rangle \}\]
\[p \ ('' \ , \ FLFun, \ seq \ (\text{str}, \ast \langle \text{str}, FLExpr \rangle)) \ \mapsto \ \{ \langle \text{str}, 1 \rangle, \langle \text{FLExpr}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr, \ FLExpr_1) \ \mapsto \ \{ \langle \text{FLExpr}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr, \ FLExpr_2) \ \mapsto \ \{ \langle \text{FLExpr}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr, \ FLExpr_3) \ \mapsto \ \{ \langle \text{FLExpr}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr, \ int) \ \mapsto \ \{ \langle \text{int}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr_1, \ seq \ (\text{FLExpr}, \ FLOp, \ FLExpr)) \ \mapsto \ \{ \langle \text{FLExpr}, 1 \rangle, \langle \text{FLExpr}, 11 \rangle \}\]
\[p \ ('' \ , \ FLExpr_2, \ seq \ (\text{str}, \ast \langle \text{FLExpr} \rangle)) \ \mapsto \ \{ \langle \text{str}, \langle \text{FLExpr}, \ast \rangle \rangle \}\]
\[p \ ('' \ , \ FLExpr_3, \ seq \ (\text{FLExpr}, \ FLExpr, \ FLExpr)) \ \mapsto \ \{ \langle \text{FLExpr}, 111 \rangle \}\]

### 7.3 Grammar in ANF

| Production rule | Production signature |
|-----------------|----------------------|
| \[p \ ('' \ , \ FLPrg, \ast \langle FLFun \rangle)\] | \{ \langle FLFun, \ast \rangle \}\]
| \[p \ ('' \ , \ FLFun, \ seq \ (\text{str}, \ast \langle \text{str}, FLExpr \rangle))\] | \{ \langle \text{str}, 1 \rangle, \langle \text{FLExpr}, 1 \rangle \}\]
| \[p \ ('' \ , \ FLExpr, \ FLExpr_1)\] | \{ \langle \text{FLExpr}, 1 \rangle \}\]
| \[p \ ('' \ , \ FLExpr, \ FLExpr_2)\] | \{ \langle \text{FLExpr}, 1 \rangle \}\]
| \[p \ ('' \ , \ FLExpr, \ FLExpr_3)\] | \{ \langle \text{FLExpr}, 1 \rangle \}\]
| \[p \ ('' \ , \ FLExpr, \ int)\] | \{ \langle \text{int}, 1 \rangle \}\]
| \[p \ ('' \ , \ FLExpr_1, \ seq \ (\text{FLExpr}, \ FLOp, \ FLExpr))\] | \{ \langle \text{FLExpr}, 1 \rangle, \langle \text{FLExpr}, 11 \rangle \}\]
| \[p \ ('' \ , \ FLExpr_2, \ seq \ (\text{str}, \ast \langle \text{FLExpr} \rangle))\] | \{ \langle \text{str}, \langle \text{FLExpr}, \ast \rangle \rangle \}\]
| \[p \ ('' \ , \ FLExpr_3, \ seq \ (\text{FLExpr}, \ FLExpr, \ FLExpr))\] | \{ \langle \text{FLExpr}, 111 \rangle \}\]

### 7.4 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

\[p \ ('' \ , \ FLPrg, \ast \langle FLFun \rangle) \ \mapsto \ \{ \langle FLFun, \ast \rangle \}\]
\[p \ ('' \ , \ FLFun, \ seq \ (\text{str}, \ast \langle \text{str}, FLExpr \rangle)) \ \mapsto \ \{ \langle \text{str}, 1 \rangle, \langle \text{FLExpr}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr, \ FLExpr_1) \ \mapsto \ \{ \langle \text{FLExpr}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr, \ FLExpr_2) \ \mapsto \ \{ \langle \text{FLExpr}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr, \ FLExpr_3) \ \mapsto \ \{ \langle \text{FLExpr}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr, \ int) \ \mapsto \ \{ \langle \text{int}, 1 \rangle \}\]
\[p \ ('' \ , \ FLExpr_1, \ seq \ (\text{FLExpr}, \ FLOp, \ FLExpr)) \ \mapsto \ \{ \langle \text{FLExpr}, 1 \rangle, \langle \text{FLExpr}, 11 \rangle \}\]
\[p \ ('' \ , \ FLExpr_2, \ seq \ (\text{str}, \ast \langle \text{FLExpr} \rangle)) \ \mapsto \ \{ \langle \text{str}, \langle \text{FLExpr}, \ast \rangle \rangle \}\]
\[p \ ('' \ , \ FLExpr_3, \ seq \ (\text{FLExpr}, \ FLExpr, \ FLExpr)) \ \mapsto \ \{ \langle \text{FLExpr}, 111 \rangle \}\]

This yields the following nominal mapping:

\[\text{rascal} - a \circ \text{master} = \{ \langle \text{FLFun, function} \rangle,\]
\[\langle \text{FLExpr}_2, \text{apply} \rangle,\]
\[\langle \text{FLPrg, program} \rangle,\]
\[\langle \text{FLExpr, expression} \rangle,\]
\[\langle \text{int, int} \rangle,\]
\[\langle \text{str, str} \rangle,\]
\[\langle \text{FLExpr}_3, \text{conditional} \rangle,\]
\[\langle \text{FLOp, operator} \rangle,\]
\[\langle \text{FLExpr}_1, \text{binary} \rangle \}\]

Which is exercised with these grammar transformation steps:

- renameN-renameN FLFun to function
- renameN-renameN FLExpr to apply
- renameN-renameN FLPrg to program
- renameN-renameN FLExpr to expression
- renameN-renameN FLExpr to conditional
• renameN-renameN \texttt{FLOp} to \textit{operator}

• renameN-renameN \texttt{FLEexpr} to \textit{binary}

7.5 Structural resolution

• \texttt{Narrow-widen} in \textit{program}
  + (\texttt{function})
  - (\texttt{function})

• \texttt{Narrow-widen} in \textit{function}
  + (\texttt{str})
  - (\texttt{str})

• \texttt{Narrow-widen} in \textit{apply}
  + (\texttt{expression})
  - (\texttt{expression})
Grammar 8

Rascal Concrete Syntax Definition

Source name: rascal-c

8.1 Source grammar

- Source artifact: topics/fl/rascal/Concrete.rsc
- Grammar extractor: shared/rascal/src/extract/RascalSyntax2BGF.rsc

| Production rules |
|------------------|
| p('prg', Program, sel('functions', s+ (Function, _))) |
| p('ifThenElse', Expr, seq ([if, sel('cond', Expr), 'then', sel('thenbranch', Expr), 'else', sel('elsebranch', Expr)])) |
| p('=', Expr, seq ([('=', sel('e', Expr), _)])) |
| p('literal', Expr, sel('i', Int)) |
| p('argument', Expr, sel('a', Name)) |
| p('binary', Expr, seq ([sel('lexpr', Expr), sel('op', Ops), sel('rexpr', Expr)])) |
| p('apply', Expr, seq ([sel('f', Name), sel('vargs', +(Expr))])) |
| p('plus', Ops, '+') |
| p('equal', Ops, '==') |
| p('minus', Ops, '-') |
| p('fun', Function, seq ([sel('f', Name), sel('args', +(Name)), '='], sel('body', Expr)])) |

8.2 Normalizations

- reroot-reroot [] to [Program]
- unlabel-designate
  p ([prar) Program, sel('functions', s+ (Function, _))]
- unlabel-designate
  p ([ifThenElse Expr, seq ([if, sel('cond', Expr), 'then', sel('thenbranch', Expr), 'else', sel('elsebranch', Expr)]))
- unlabel-designate
  p ([literal Expr, sel('i', Int))
- unlabel-designate
  p ([argument Expr, sel('a', Name))
- unlabel-designate
  p ([binary Expr, seq ([sel('lexpr', Expr), sel('op', Ops), sel('rexpr', Expr)]))
- unlabel-designate
  p ([apply Expr, seq ([sel('f', Name), sel('vargs', +(Expr)]))]
- unlabel-designate
  p ([plus Ops, '+')]
• unlabel-designate
  p (equal, Ops, '==')
• unlabel-designate
  p (minus, Ops, '-')
• unlabel-designate
  p (fun, Function, seq ([sel ('f', Name), sel ('args', + (Name))), '==', sel ('body', Expr)])
• anonymize-deanonymize
  p ('', Expr, seq ([sel ('f', Name), sel ('args', + (Expr))), '==', sel ('body', Expr)])
• anonymize-deanonymize
  p ('', Function, seq ([sel ('f', Name), sel ('args', + (Name))), '==', sel ('body', Expr)])
• anonymize-deanonymize
  p ('', Expr, seq ([if, sel ('cond', Expr), 'then', sel ('thenbranch', Expr), 'else', sel ('elsebranch', Expr)])
• anonymize-deanonymize
  p ('', Expr, seq ([if, sel ('e', Expr), 'then', Expr, 'else', Expr])
• abstractize-concretize
  p ('', Expr, seq ([Expr, Expr, Expr])
• abstractize-concretize
  p ('', Function, seq ([Name, + (Expr)])
• abstractize-concretize
  p ('', Ops)
• abstractize-concretize
  p ('', Expr, Int)
• abstractize-concretize
  p ('functions', Program, s+ (Function))
• abstractize-concretize
  p ('', Expr, seq ([Expr, Expr, then, Expr, else, Expr])
• undefine-define
  p ('', Ops, ε)
• abridge-detour
  p ('', Expr, Expr)
• unlabel-designate
  p ('functions', Program, + (Function))
• unlabel-designate
  p ('', Expr, Int)
• unlabel-designate
  p ('', Expr, Name)
• extract-inline in Expr
  p ('', Expr, seq ([Expr, Expr, Expr])
• extract-inline in Expr
  p ('', Expr, seq ([Expr, Ops, Expr])
• extract-inline in Expr
  p ('', Expr, seq ([Name, + (Expr)])

31
8.3 Grammar in ANF

| Production rule | Production signature          |
|-----------------|------------------------------|
| \( p(\cdot, \text{Program}, +(\text{Function})) \) | \{ \{ \text{Function}, + \} \} |
| \( p(\cdot, \text{Expr}, \text{Expr}_2) \)            | \{ \{ \text{Expr}, 1 \} \} |
| \( p(\cdot, \text{Expr}, \text{Int}) \)             | \{ \{ \text{Int}, 1 \} \} |
| \( p(\cdot, \text{Expr}, \text{Name}) \)            | \{ \{ \text{Name}, 1 \} \} |
| \( p(\cdot, \text{Expr}, \text{Expr}_2) \)            | \{ \{ \text{Expr}, 2 \} \} |
| \( p(\cdot, \text{Expr}, \text{Expr}_3) \)            | \{ \{ \text{Expr}, 1 \} \} |
| \( p(\cdot, \text{Function}, \text{seq}(\{\text{Name},+(\text{Name}), \text{Expr}\})) \) | \{ \{ \text{Expr}, 1 \}, \{ \text{Name}, 1 \} \} |
| \( p(\cdot, \text{Expr}_1, \text{seq}(\{\text{Expr}, \text{Expr}, \text{Expr}\})) \) | \{ \{ \text{Expr}, 1 \} \} |
| \( p(\cdot, \text{Expr}_2, \text{seq}(\{\text{Expr}, \text{Ops}, \text{Expr}\})) \) | \{ \{ \text{Ops}, 1 \}, \{ \text{Expr}, 1 \} \} |
| \( p(\cdot, \text{Expr}_3, \text{seq}(\{\text{Name},+(\text{Expr})\})) \) | \{ \{ \text{Expr}, + \}, \{ \text{Name}, 1 \} \} |

8.4 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

- \( p(\cdot, \text{Program}, +(\text{Function})) \) \( \rightarrow \) \( p(\cdot, \text{program}, +(\text{function})) \)
- \( p(\cdot, \text{Expr}, \text{Expr}_1) \) \( \rightarrow \) \( p(\cdot, \text{expression}, \text{conditional}) \)
- \( p(\cdot, \text{Expr}, \text{Int}) \) \( \rightarrow \) \( p(\cdot, \text{expression}, \text{int}) \)
- \( p(\cdot, \text{Expr}, \text{Name}) \) \( \rightarrow \) \( p(\cdot, \text{expression}, \text{str}) \)
- \( p(\cdot, \text{Expr}, \text{Expr}_2) \) \( \rightarrow \) \( p(\cdot, \text{expression}, \text{binary}) \)
- \( p(\cdot, \text{Expr}, \text{Expr}_3) \) \( \rightarrow \) \( p(\cdot, \text{expression}, \text{apply}) \)
- \( p(\cdot, \text{Function}, \text{seq}(\{\text{Name},+(\text{Name}), \text{Expr}\})) \) \( \rightarrow \) \( p(\cdot, \text{function}, \text{seq}(\{\text{str}, +(\text{str}), \text{expression}\})) \)
- \( p(\cdot, \text{Expr}_1, \text{seq}(\{\text{Expr}, \text{Expr}, \text{Expr}\})) \) \( \rightarrow \) \( p(\cdot, \text{conditional}, \text{seq}(\{\text{expression}, \text{expression}, \text{expression}\})) \)
- \( p(\cdot, \text{Expr}_2, \text{seq}(\{\text{Expr}, \text{Ops}, \text{Expr}\})) \) \( \rightarrow \) \( p(\cdot, \text{binary}, \text{seq}(\{\text{expression}, \text{operator}, \text{expression}\})) \)
- \( p(\cdot, \text{Expr}_3, \text{seq}(\{\text{Name}, +(\text{Expr})\})) \) \( \rightarrow \) \( p(\cdot, \text{apply}, \text{seq}(\{\text{str}, +(\text{expression})\})) \)

This yields the following nominal mapping:

\[
\text{rascal} - c \circ \text{master} = \{ \{\text{Expr}_2, \text{binary}\}, \\
\quad \{\text{Int}, \text{int}\}, \\
\quad \{\text{Expr}_1, \text{conditional}\}, \\
\quad \{\text{Function}, \text{function}\}, \\
\quad \{\text{Program}, \text{program}\}, \\
\quad \{\text{Name}, \text{str}\}, \\
\quad \{\text{Expr}_3, \text{apply}\}, \\
\quad \{\text{Expr}, \text{expression}\}, \\
\quad \{\text{Ops}, \text{operator}\}\}
\]

Which is exercised with these grammar transformation steps:

- renameN-renameN \( \text{Expr}_2 \) to \( \text{binary} \)
- renameN-renameN \( \text{Int} \) to \( \text{int} \)
- renameN-renameN \( \text{Expr}_1 \) to \( \text{conditional} \)
- renameN-renameN \( \text{Function} \) to \( \text{function} \)
- renameN-renameN \( \text{Program} \) to \( \text{program} \)
- renameN-renameN \( \text{Name} \) to \( \text{str} \)
- renameN-renameN \( \text{Expr}_3 \) to \( \text{apply} \)
- renameN-renameN \( \text{Expr} \) to \( \text{expression} \)
- renameN-renameN \( \text{Ops} \) to \( \text{operator} \)
Grammar 9

Syntax Definition

Formalism

Source name: sdf

9.1 Source grammar

- Source artifact: topics/fl/asfsdf/Syntax.sdf
- Grammar extractor: topics/extraction/sdf/Main.sdf
- Grammar extractor: topics/extraction/sdf/Main.asf
- Grammar extractor: topics/extraction/sdf/Tokens.sdf
- Grammar extractor: topics/extraction/sdf/Tokens.asf

| Production rules |
|------------------|
| $p(\epsilon, \text{Program}, +(\text{Function}))$ |
| $p(\epsilon, \text{Function}, \text{seq} ([\text{Name}, +(\text{Name}), \text{\textbf{=}}, \text{Expr}, +(\text{Newline})]))$ |
| $p\text{\textbf{binary}}, \text{Expr}, \text{seq} ([\text{Expr}, \text{Ops}, \text{Expr}])$ |
| $p\text{\textbf{apply}}, \text{Expr}, \text{seq} ([\text{Name}, +(\text{Expr})])$ |
| $p\text{\textbf{ifThenElse}}, \text{Expr}, \text{seq} ([\text{if}, \text{Expr}, \text{then}, \text{Expr}, \text{else}, \text{Expr}])$ |
| $p(\text{argument}, \text{Expr}, \text{Name})$ |
| $p\text{\textbf{literal}}, \text{Expr}, \text{Int}$ |
| $p\text{\textbf{minus}}, \text{Ops}, \text{\textbf{-}}$ |
| $p\text{\textbf{plus}}, \text{Ops}, \text{\textbf{+}}$ |
| $p\text{\textbf{equal}}, \text{Ops}, \text{\textbf{==}}$ |

9.2 Normalizations

- reroot-reroot $[] \rightarrow [\text{Program}]$
- unlabel-designate
  - $p\text{\textbf{binary}}, \text{Expr}, \text{seq} ([\text{Expr}, \text{Ops}, \text{Expr}])$
- unlabel-designate
  - $p\text{\textbf{apply}}, \text{Expr}, \text{seq} ([\text{Name}, +(\text{Expr})])$
- unlabel-designate
  - $p\text{\textbf{ifThenElse}}, \text{Expr}, \text{seq} ([\text{if}, \text{Expr}, \text{then}, \text{Expr}, \text{else}, \text{Expr}])$
- unlabel-designate
  - $p\text{\textbf{argument}}, \text{Expr}, \text{Name}$
- unlabel-designate
  - $p\text{\textbf{literal}}, \text{Expr}, \text{Int}$
9.3 Grammar in ANF

| Production rule | Production signature |
|-----------------|----------------------|
| p (', Program, +(Function))  | {(Function, +)}       |
| p (', Function, seq ((Name, +(Name), Expr, +(Newline)))) | {(Expr, 1), (Newline, +), (Name, 1+)} |
| p (', Expr, Expr₁) | {(Expr₁, 1)}         |
| p (', Expr, Expr₁) | {(Expr₂, 1)}         |
| p (', Expr, Expr₁) | {(Expr₁, 1)}         |
| p (', Expr, Name) | {(Name, 1)}          |
| p (', Expr, Int) | {Int, 1}             |
| p (', Expr₁, seq ([Expr, Ops, Expr₃])) | {(Ops, 1), (Expr, 11)} |
| p (', Expr₂, seq ((Name, +(Expr)))) | {(Expr, +), (Name, 1)} |
| p (', Expr₃, seq ([Expr, Expr, Expr₃])) | {(Expr, 111)}        |
9.4 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

\[ p('', \text{Program}, +(\text{Function})) \cong p('', \text{program}, +(\text{function})) \]
\[ p('', \text{Function}, \text{seq}([\text{Name}, +\text{Name}], \text{Expr}, +(\text{Newline}))) \cong p('', \text{function}, \text{seq}([\text{str}, +(\text{str}), \text{expression}]))) \]
\[ p('', \text{Expr}, \text{Expr}_1) \cong p('', \text{expression}, \text{binary}) \]
\[ p('', \text{Expr}, \text{Expr}_2) \cong p('', \text{expression}, \text{apply}) \]
\[ p('', \text{Expr}, \text{Expr}_3) \cong p('', \text{expression}, \text{conditional}) \]
\[ p('', \text{Expr}, \text{Name}) \cong p('', \text{expression}, \text{str}) \]
\[ p('', \text{Expr}, \text{Int}) \cong p('', \text{expression}, \text{int}) \]
\[ p('', \text{Expr}_1, \text{seq}([\text{Expr}, \text{Ops}, \text{Expr}])) \cong p('', \text{binary}, \text{seq}([\text{expression}, \text{operator}, \text{expression}])) \]
\[ p('', \text{Expr}_2, \text{seq}([\text{Name}, +(\text{Expr})])) \cong p('', \text{apply}, \text{seq}([\text{str}, +(\text{expression})))) \]
\[ p('', \text{Expr}_3, \text{seq}([\text{Expr}, \text{Expr}, \text{Expr}])) \cong p('', \text{conditional}, \text{seq}([\text{expression}, \text{expression}, \text{expression}])) \]

This yields the following nominal mapping:

\[ \text{sdf} \circ \text{master} = \{ (\text{Expr}_3, \text{conditional}), \]
\[ (\text{Int}, \text{int}), \]
\[ (\text{Expr}_1, \text{binary}), \]
\[ (\text{Newline}, \omega), \]
\[ (\text{Function}, \text{function}), \]
\[ (\text{Program}, \text{program}), \]
\[ (\text{Name}, \text{str}), \]
\[ (\text{Expr}, \text{expression}), \]
\[ (\text{Ops}, \text{operator}), \]
\[ (\text{Expr}_2, \text{apply}) \} \]

Which is exercised with these grammar transformation steps:

- renameN-renameN \text{Expr}_3 to conditional
- renameN-renameN \text{Int} to int
- renameN-renameN \text{Expr}_1 to binary
- renameN-renameN \text{Function} to function
- renameN-renameN \text{Program} to program
- renameN-renameN \text{Name} to str
- renameN-renameN \text{Expr} to expression
- renameN-renameN \text{Ops} to operator
- renameN-renameN \text{Expr}_2 to apply

9.5 Structural resolution

- project-inject
  \[ p('', \text{function}, \text{seq}([\text{str}, +(\text{str}), \text{expression}, +(\text{Newline})))) \]
Grammar 10

TXL

Source name: txl

10.1 Source grammar

- Source artifact: topics/fl/txl/FL.Txl
- Grammar extractor: topics/extraction/txl/txl2bgf.xslt

| Production rules |
|--------------------|
| p(‘’, program, +{fun}) |
| p(‘’, fun, seq ([(id, +(id), ‘=’, expression, newline)]) |
| p(‘’, expression, choice([seq ([(expression, op, expression)]) ,
                            seq ([(id, +(expression)])]) ,
                       seq ([(‘if’, expression, ‘then’, expression, ‘else’, expression)]) ,
                       id, number]) |
| p(‘’, op, choice([‘+’, ‘-’, ‘==’])) |

10.2 Normalizations

- abstractize-concretize
  p (‘’, fun, seq ([(id, +(id), expression, newline)]) )

- abstractize-concretize
  p (‘’, op, choice([‘+’, ‘-’, ‘==’]))

- vertical-horizontal in expression
  p (‘’, expression, choice ([seq (expression, op, expression)]) , seq ([(id, +(expression)])], seq ([(‘if’, expression, ‘then’, expression, ‘else’, expression)]) ,
                       id, number))

- undefined-define
  p (‘’, op, ε)

- abridge-detour
  p (‘’, expression, expression)

- extract-inline in expression
  p (‘’, expression, seq ([(expression, op, expression)]) )

- extract-inline in expression
  p (‘’, expression, seq ([(id, +(expression)]) )

- extract-inline in expression
  p (‘’, expression, expression, seq ([(expression, op, expression)]) )

36
10.3 Grammar in ANF

| Production rule | Production signature |
|-----------------|----------------------|
| \(p('\text{c}, \text{program, -(fun)})\) | \{\{\text{fun}, +\}\} |
| \(p('\text{c}, \text{fun, seq ([id, +(id), expression, newline])})\) | \{(\text{newline}, 1), (\text{id, 1}+), (\text{expression}, 1)\} |
| \(p('\text{c}, \text{expression, expression})\) | \{(\text{expression}, 1)\} |
| \(p('\text{c}, \text{expression, expression}_2)\) | \{(\text{expression}_2, 1)\} |
| \(p('\text{c}, \text{expression, expression}_3)\) | \{(\text{expression}_3, 1)\} |
| \(p('\text{c}, \text{expression, id})\) | \{(\text{id}, 1)\} |
| \(p('\text{c}, \text{expression, number})\) | \{(\text{number, 1})\} |
| \(p('\text{c}, \text{expression}_1, \text{seq ([expression, op, expression])})\) | \{(\text{op, 1}), (\text{expression}, 11)\} |
| \(p('\text{c}, \text{expression}_2, \text{seq ([id, +(expression)])})\) | \{(\text{expression} +), (\text{id}, 1)\} |
| \(p('\text{c}, \text{expression}_3, \text{seq ([expression, expression, expression])})\) | \{(\text{expression}, 111)\} |

10.4 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

- \(p('\text{c}, \text{program, +(fun)})\) \(\equiv\) \(p('\text{c}, \text{program, +(function)})\)
- \(p('\text{c}, \text{fun, seq ([id, +(id), expression, newline])})\) \(\equiv\) \(p('\text{c}, \text{fun, seq ([str, +(str), expression])})\)
- \(p('\text{c}, \text{expression, expression}_1)\) \(\equiv\) \(p('\text{c}, \text{expression, binary})\)
- \(p('\text{c}, \text{expression, expression}_2)\) \(\equiv\) \(p('\text{c}, \text{expression, apply})\)
- \(p('\text{c}, \text{expression, expression}_3)\) \(\equiv\) \(p('\text{c}, \text{expression, conditional})\)
- \(p('\text{c}, \text{expression, id})\) \(\equiv\) \(p('\text{c}, \text{expression, str})\)
- \(p('\text{c}, \text{expression, number})\) \(\equiv\) \(p('\text{c}, \text{expression, int})\)

This yields the following nominal mapping:

\[\text{tsl} \circ \text{master} = \{(\text{program, program}), (\text{expression}_2, \text{apply}), (\text{fun, function}), (\text{expression, expression}), (\text{id, str}), (\text{expression}_1, \text{binary}), (\text{op, operator}), (\text{number, int}), (\text{newline, \omega}), (\text{expression}_3, \text{conditional})\}\]

Which is exercised with these grammar transformation steps:

- renameN-renameN expression\(_2\) to apply
- renameN-renameN fun to function
- renameN-renameN id to str
- renameN-renameN expression\(_1\) to binary
- renameN-renameN op to operator
- renameN-renameN number to int
- renameN-renameN expression\(_3\) to conditional
10.5 Structural resolution

- **project-inject**
  
  \[
  p \left( ^{1} , \text{function} , \text{seq} \left( \left[ \text{str} , \text{str} , \text{expression}, \text{newline} \right] \right) \right)
  \]
Grammar 11

XML Schema

Source name: xsd

11.1 Source grammar

- Source artifact: topics/fl/xsd/fl.xsd
- Grammar extractor: shared/prolog/xsd2bgf.pro

| Production rules |
|------------------|
| `p("", Program, +\((sel\(\"function\", Function))\))` |
| `p("", Fragment, Expr)` |
| `p("", Function, seq\([sel\(\"name\", str\), +\((sel\(\"arg\", str\)), sel\(\"rhs\", Expr))\])` |
| `p("", Expr, choice\([Literal, Argument, Binary, IfThenElse, Apply])\))` |
| `p("", Literal, sel\(\"info\", int\))` |
| `p("", Argument, sel\(\"name\", str\))` |
| `p("", Binary, seq\([sel\(\"ops\", Ops\), sel\(\"left\", Expr\), sel\(\"right\", Expr))\])` |
| `p("", Ops, choice\([sel\(\"Equal\", ε\), sel\(\"Plus\", ε\), sel\(\"Minus\", ε\)])` |
| `p("", IfThenElse, seq\([sel\(\"ifExpr\", Expr\), sel\(\"thenExpr\", Expr\), sel\(\"elseExpr\", Expr\)])` |
| `p("", Apply, seq\([sel\(\"name\", str\), +\((sel\(\"arg\", Expr))\])\))` |

11.2 Normalizations

- unlabeled-designate
  `p\(\"\", Literal, int)\)`

- unlabeled-designate
  `p\(\"\", Argument, str\)`

- anonymize-deanonymize
  `p\(\"\", Apply, seq\([sel\(\"name\", str\), +\((sel\(\"arg\", Expr))\)])\)`

- anonymize-deanonymize
  `p\(\"\", Function, seq\([sel\(\"name\", str\), +\((sel\(\"arg\", str\)), sel\(\"rhs\", Expr))\)])`

- anonymize-deanonymize
  `p\(\"\", IfThenElse, seq\([sel\(\"ifExpr\", Expr\), sel\(\"thenExpr\", Expr\), sel\(\"elseExpr\", Expr\)])\)`

- anonymize-deanonymize
  `p\(\"\", Program, +\((sel\(\"function\", Function))\))`
• anonymize-deanonymize
  p ('', Ops, choice (sel ('Equal', ε), sel ('Plus', ε), sel ('Minus', ε)))

• anonymize-deanonymize
  p ('', Binary, seq (sel ('ops', Ops), sel ('left', Expr), sel ('right', Expr)))

• vertical-horizontal in Expr

• undefine-define
  p ('', Ops, ε)

• unchain-chain
  p ('', Expr, Literal)

• unchain-chain
  p ('', Expr, Argument)

• unchain-chain
  p ('', Expr, Binary)

• unchain-chain
  p ('', Expr, IfThenElse)

• unchain-chain
  p ('', Expr, Apply)

• unlabel-designate
  p ('Literal', Expr, int)

• unlabel-designate
  p ('Argument', Expr, str)

• unlabel-designate
  p ('Binary', Expr, seq ([Ops, Expr, Expr]))

• unlabel-designate
  p ('IfThenElse', Expr, seq ([Expr, Expr, Expr]))

• unlabel-designate
  p ('Apply', Expr, seq ([str, +(Expr)]))

• extract-inline in Expr
  p ('', Expr, seq ([str, +(Expr)]))

• extract-inline in Expr
  p ('', Expr, seq ([str, +(Expr)]))

• extract-inline in Expr
  p ('', Expr, seq ([str, +(Expr)]))

11.3 Grammar in ANF

| Production rule          | Production signature |
|--------------------------|----------------------|
| p ('', Program, +(Function)) | {⟨Function, +⟩}          |
| p ('', Fragment, Expr)    | {⟨Expr, 1⟩}           |
| p ('', Function, seq ([str, +(str), Expr])) | {⟨str, 1⟩, ⟨Expr, 1⟩} |
| p ('', Expr, int)        | {⟨int, 1⟩}            |
| p ('', Expr, str)        | {⟨str, 1⟩}             |
| p ('', Expr, Expr1)      | {⟨Expr1, 1⟩}          |
| p ('', Expr, Expr2)      | {⟨Expr2, 1⟩}          |
| p ('', Expr, Expr3)      | {⟨Expr3, 1⟩}          |
| p ('', Expr, seq ([Ops, Expr, Expr])) | {⟨Ops, 1⟩, ⟨Expr, 11⟩} |
| p ('', Expr, seq ([Expr, Expr, Expr])) | {⟨Expr, 111⟩}          |
| p ('', Expr, seq ([str, +(Expr)])) | {⟨str, 1⟩, ⟨Expr, +⟩} |
11.4 Nominal resolution

Production rules are matched as follows (ANF on the left, master grammar on the right):

\[
\begin{align*}
p\left(\"\cdot\", \text{Program}, +\left(\text{Function}\right)\right) & \equiv p\left(\"\cdot\", \text{program}, +\left(\text{function}\right)\right) \\
p\left(\"\cdot\", \text{Fragment, Expr}\right) & \equiv \emptyset \\
p\left(\"\cdot\", \text{Function, seq}\left(\left[\text{str, +\left(\text{str} \right)}, \text{Expr}\right]\right)\right) & \equiv p\left(\"\cdot\", \text{function, seq}\left(\left[\text{str, +\left(\text{str} \right)}, \text{expression}\right]\right)\right) \\
p\left(\"\cdot\", \text{Expr, int}\right) & \equiv p\left(\"\cdot\", \text{expression, int}\right) \\
p\left(\"\cdot\", \text{Expr, str}\right) & \equiv p\left(\"\cdot\", \text{expression, str}\right) \\
p\left(\"\cdot\", \text{Expr, Expr}\right) & \equiv p\left(\"\cdot\", \text{expression, conditional}\right) \\
p\left(\"\cdot\", \text{Expr, Expr1}, \text{Expr2}\right) & \equiv p\left(\"\cdot\", \text{expression, apply}\right) \\
p\left(\"\cdot\", \text{Expr1, seq}\left(\left[\text{Ops, Expr, Expr}\right]\right)\right) & \equiv p\left(\"\cdot\", \text{binary, seq}\left(\left[\text{expression, operator, expression}\right]\right)\right) \\
p\left(\"\cdot\", \text{Expr2, seq}\left(\left[\text{Expr, Expr, Expr}\right]\right)\right) & \equiv p\left(\"\cdot\", \text{conditional, seq}\left(\left[\text{expression, expression, expression}\right]\right)\right) \\
p\left(\"\cdot\", \text{Expr3, seq}\left(\left[\text{str, +\left(\text{Expr}\right)}\right]\right)\right) & \equiv p\left(\"\cdot\", \text{apply, seq}\left(\left[\text{str, +\left(\text{expression}\right)}\right]\right)\right)
\end{align*}
\]

This yields the following nominal mapping:

\[
\text{xsd} \odot \text{master} = \{\left(\text{Expr1, binary}\right), \\
\left(\text{str, str}\right), \\
\left(\text{int, int}\right), \\
\left(\text{Expr2, conditional}\right), \\
\left(\text{Function, function}\right), \\
\left(\text{Program, program}\right), \\
\left(\text{Expr3, apply}\right), \\
\left(\text{Expr, expression}\right), \\
\left(\text{Ops, operator}\right)\}
\]

Which is exercised with these grammar transformation steps:

- renameN-renameN \text{Expr1} to \text{binary}
- renameN-renameN \text{Expr2} to \text{conditional}
- renameN-renameN \text{Function} to \text{function}
- renameN-renameN \text{Program} to \text{program}
- renameN-renameN \text{Expr3} to \text{apply}
- renameN-renameN \text{Expr} to \text{expression}
- renameN-renameN Ops to operator

11.5 Structural resolution

- reroot-reroot \{\text{program, Fragment}\} to \{\text{program}\}
- eliminate-introduce
  \[p\left(\"\cdot\", \text{Fragment, expression}\right)\]
- permute-permute
  \[p\left(\"\cdot\", \text{binary, seq}\left(\left[\text{operator, expression, expression}\right]\right)\right)\]
  \[p\left(\"\cdot\", \text{binary, seq}\left(\left[\text{expression, operator, expression}\right]\right)\right)\]
Bibliography

[1] J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. OOPSLA & GPCE, Workshop on best MDSD practices, 2004. Publicly available via http://www.softmetaware.com/oopsla2004/bezivin-megamodel.pdf.

[2] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. Grammar Programming in TXL. In Proceedings of SCAM 2002. IEEE, 2002. Publicly available via http://plg1.uwaterloo.ca/~ajmalton/ajmalton/Papers/SCAM02_GP.pdf.

[3] Eclipse. Eclipse Modeling Framework Project (EMF 2.4), 2008. http://www.eclipse.org/modeling/emf/.

[4] G. Economopoulos, P. Klint, and J. J. Vinju. Faster scannerless GLR parsing. In O. de Moor and M. I. Schwartzbach, editors, Proceedings of CC 2009, volume 5501 of LNCS, pages 126–141. Springer, 2009. Publicly available via http://oai.cwi.nl/oai/asset/15095/15095B.pdf.

[5] J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In Proceedings of MODELS 2012, LNCS. Springer, 2012.

[6] J.-M. Favre and T. NGuen. Towards a Megamodel to Model Software Evolution through Transformations. In Proceedings of SETra, volume 127 of ENTCS, 2004. Publicly available via http://adele.imag.fr/Les.Publications/intConferences/SETRAa2004Fav.pdf.

[7] J. Fialli and S. Vajjhala. Java Specification Request 31: XML Data Binding Specification, 1999. Publicly available via http://jcp.org/en/jsr/detail?id=031.

[8] S. Gao, C. M. Sperberg-McQueen, and H. S. Thompson. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. W3C Recommendation, Apr. 2012. Publicly available via http://www.w3.org/TR/2012/REC-xmleschma1-1-20120405.

[9] P. Klint. A Meta-Environment for Generating Programming Environments. ACM TOSEM, 2(2):176–201, 1993. Publicly available via http://dare.uva.nl/document/28101.

[10] P. Klint et al. Rascal Tutor. SWAT, CWI, 2012. http://tutor.rascal-mpl.org.

[11] P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, Post-proceedings of GTTSE 2009, volume 6491 of LNCS, pages 222–289. Springer-Verlag, January 2011. Publicly available via http://homepages.cwi.nl/~paulk/publications/rascal-gttse-final.pdf.

[12] R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In M. Leuschel and H. Wehrheim, editors, Proceedings of iFM 2009, volume 5423 of LNCS, pages 246–260. Springer-Verlag, February 2009. Publicly available via http://grammarware.net/writes#Convergence2009.

[13] P. McGuire. Getting Started with PyParsing. O’Reilly, first edition, 2007.

[14] Object Management Group. Meta-Object Facility (MOF™) Core Specification, 2.0 edition, Jan. 2006. Publicly available via http://www.omg.org/spec/MOF/2.0.
[15] T. Parr. ANTLR—ANother Tool for Language Recognition, 2008. http://antlr.org.

[16] F. C. N. Pereira and D. H. D. Warren. Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks. Artificial Intelligence, 13:231–278, 1980. Publicly available via http://cgi.di.uoa.gr/~takis/pereira-warren.pdf.

[17] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, Universiteit van Amsterdam, July 1997. Publicly available via http://www.science.uva.nl/pub/programming-research/reports/1997/P9707.ps.Z.

[18] V. Zaytsev. Guided Grammar Convergence. Submitted to POPL 2013. Pending notification. June 2012. Publicly available via http://grammarware.net/writes#Guided2013.

[19] V. Zaytsev. Renarrating Linguistic Architecture: A Case Study. Submitted to MPM 2012. Pending notification. July 2012. Publicly available via http://grammarware.net/writes#Renarration2012.

[20] V. Zaytsev. Language Evolution, Metasyntactically. In Proceedings of BX 2012, volume 49 of EC-EASST. EASST, 2012. Publicly available via http://grammarware.net/writes#Metasyntactically2012.

[21] V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, and G. Wachsmuth. Software Language Processing Suite\textsuperscript{1}, 2008–2012. http://grammarware.github.com.

\textsuperscript{1}The authors are given according to the statistics at http://github.com/grammarware/slps/graphs/contributors.