Island Grammar-based Parsing using GLL and Tom

Ali Afroozeh, Jean-Christophe Bach, Mark Van den Brand, Adrian Johnstone, Maarten Manders, Pierre-Etienne Moreau, Elizabeth Scott

Island Grammar-based Parsing using GLL and Tom

Ali Afroozeh[1], Jean-Christophe Bach[2,3,4], Mark van den Brand[1], Adrian Johnstone[5], Maarten Manders[1], Pierre-Etienne Moreau[2,3,4], and Elizabeth Scott[5]

[1] Eindhoven University of Technology, NL-5612 AZ Eindhoven, The Netherlands
afroozeh@gmail.com, m.g.j.v.d.brand@tue.nl, m.w.manders@tue.nl
[2] Inria, Villers-lès-Nancy, F-54600, France
[3] Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France
[4] CNRS, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France
jeanchristophe.bach@inria.fr, pierre-etienne.moreau@loria.fr
[5] Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK
a.johnstone@rhul.ac.uk, eas@cs.rhul.ac.uk

Abstract. Extending a language by embedding within it another language presents significant parsing challenges, especially if the embedding is recursive. The composite grammar is likely to be nondeterministic as a result of tokens that are valid in both the host and the embedded language. In this paper we examine the challenges of embedding the Tom language into a variety of general-purpose high level languages. Tom provides syntax and semantics for advanced pattern matching and tree rewriting facilities. Embedded Tom constructs are translated into the host language by a preprocessor, the output of which is a composite program written purely in the host language. Tom implementations exist for Java, C, C#, Python and Caml. The current parser is complex and difficult to maintain. In this paper, we describe how Tom can be parsed using island grammars implemented with the Generalised LL (GLL) parsing algorithm. The grammar is, as might be expected, ambiguous. Extracting the correct derivation relies on our disambiguation strategy which is based on pattern matching within the parse forest. We describe different classes of ambiguity and propose patterns for resolving them.

Keywords: GLL, Tom, island grammars, parsing, disambiguation

1 Introduction

Modern software systems are composed of a wide range of programming languages. In many cases there is even a mixture of programming languages within one program. A traditional example is Cobol with CICS or SQL embeddings. It is possible for these embeddings to be realised via strings in which the extension constructs are encoded. In this case the parser of the host language does not need to be aware of the fact that another language is present. The drawback of

these string encodings is that syntax errors in the embedded language constructs are not detected until a later phase in which these constructs are processed.

In the last decade language developers have been working on extending general-purpose programming languages with domain-specific languages, referred to in this paper as guest languages. The language embeddings which are not simple string encodings present a challenge from a parsing point of view, especially if the general-purpose programming language and the embeddings can be unboundedly nested. Tom [1] is an example of such an extension which provides general-purpose programming languages, such as Java, C, C#, Python, and Caml, referred to in this paper as the host language, with advanced pattern matching, term rewriting and tree traversal functionality. The Tom compiler processes the Tom constructs and generates corresponding host language code. The host-language compiler can then be used to build the final program from the generated code. The main advantage of this two-phase compiling approach is that the host-language compiler remains unaware of the extension constructs. As a result, the host language can evolve without breaking the extension's compiler, and the guest language can be used to extend other host languages.

If language embeddings are not simple string encodings, the syntax of the host language needs to be modified to accommodate the guest language. This is usually done by using tags which signal the beginning and end of guest constructs. In case of nesting host constructs inside guest construct, another set of tags may be employed for notifying the parser of the return to the host language [2]. However, the use of tags for switching between languages is not very convenient for developers who use the language. In Tom, only an opening tag is used, while the closing tag is the same as the closing tag of blocks in the host language. Furthermore, switching to the host language inside Tom constructs does not need any special tag, and the host and guest languages may be unboundedly nested. These features make parsing Tom even more challenging.

Parsing the full syntax of the host language is neither desirable nor practical in many cases, especially for Tom, in which we only need to extract the guest constructs. One way to avoid parsing the full syntax of the host language is to use island grammars [3]. An island grammar captures the important constructs, embedded constructs in our case, as "island" and ignores the rest as "water". Two main issues should be taken into consideration while parsing island grammars. First, the class of deterministic context-free languages is not closed under union, meaning that the union of two deterministic languages may no longer be deterministic. Therefore, even if one designs LL(k) or LR(k) syntax for a language extension, there is no guarantee that the resulting language is in the same class. Second, the host and extension languages may share tokens, which may lead to ambiguities. The ambiguities from shared tokens cannot always be resolved by rewriting the grammar, using more lookahead tokens or backtracking, so there is a need for more sophisticated disambiguation schemes.

Attempting to parse an island grammar of a language using standard LL or LR parsing techniques will, at the very least, involve significant modifications in the parser and, in worst case, may not even be possible. For example, the current Tom parser uses multiple parsers, implemented in ANTLR[1], to deal with the host and Tom constructs separately. This implementation is complex, hard to maintain, and does not reflect the grammar of the language. Moreover, having a complex parser implementation makes the changes in and evolution of (both) languages a burden.

During the last 30 years, more efficient implementations of generalised parsers have become available. Since the algorithm formulated by Tomita [4] there have been a number of generalised LR parsing (GLR) implementations, such as GLR [5], a scannerless variant (SGLR) [6] and Dparser[2]. Johnstone and Scott developed a generalised LL (GLL) parser [7] in which the function call stack in a traditional recursive descent parser is replaced by a structure similar to the stack data structure (Graph Structured Stack) used in GLR parsing. GLL parsers are particularly interesting because their implementations are straightforward and efficient.

Our goal is to avoid manipulation within a parser in order to provide a generic, reusable solution for parsing language embeddings. We have chosen Tom as our case study, mainly because of challenges involved in parsing Tom such as recursive embedding and the lack of closing tags. In this paper, we present an island grammar for Tom in EBNF. The fundamental question is how to deal with ambiguities present in island grammars which support recursive embedding. For disambiguation we perform pattern matching within the parse forest. To validate our approach, we conducted parsing experiments using an improved version of our Java implementation of GLL [8].

The rest of this paper is organised as follows. In Sect. 2 we introduce Tom as a language for term rewriting and give a brief description of the GLL parsing algorithm along with some notes on our GLL implementation. Section 3 describes the idea of island grammars by defining an island grammar for Tom. In Sect. 4 we illustrate our mechanism for resolving ambiguities in island-based grammars by providing disambiguation rules for different types of ambiguities present in Tom. The results of parsing Tom examples are presented in Sect. 5. In Sect. 6 we present other work in the area of parsing embedded languages and compare our approach with them. Finally, in Sect. 7, a conclusion to this paper and some ideas for future work are given.

2 Preliminaries

In this section we introduce the Tom language which is used for two different purposes in the rest of this paper. First, Tom is used as an example of an em-

[1] `http://www.antlr.org/`
[2] `http://dparser.sourceforge.net/`

bedded syntax with recursive nesting, which poses difficulties for conventional deterministic parsers. Second, Tom is used as a pattern matching and rewriting technology for implementing a disambiguation mechanism for island grammars. The rest of this section gives a brief explanation of the GLL parsing algorithm and our Java-based GLL implementation.

2.1 Tom in a Nutshell

Tom [9,1] is a language based on rewriting calculus and is designed to integrate term rewriting and pattern matching facilities into general-purpose programming languages (GPLs) such as Java, C, C#, Python, and Caml. Tom relies on the Formal Island framework [10], meaning that the underlying host language does not need to be parsed in order to compile Tom constructs.

The basic functionality of Tom is pattern matching through the `%match` construct. This construct can be seen as a generalisation of the *switch-case* construct in many GPLs. The `%match` construct is composed of a set of rules where the left-hand side is a *pattern*, *i.e.*, a tree that may contain variables, and the right-hand side an *action*, given by a Java block that may in turn contain Tom constructs.

A second construct provided by Tom is the backquote (`‘`) term. Given an algebraic term, *i.e.*, a tree, a backquote term builds the corresponding data structure by allocating and initialising the required objects in memory.

A third component of the Tom language is a formalism to describe algebraic data structures by means of the `%gom` construct. This corresponds to the definition of inductive types in classical functional programming. There are two main ways of using this formalism: the first one is defining an algebraic data type in Gom and generating Java classes which implement the data type. This is similar to the Eclipse Modeling Framework[3], in which a Java implementation can be generated from a meta-model definition. The second approach assumes that a data structure implementation, for example in Java, already exists. Then, an algebraic data type in Gom can be defined to provide a *mapping* to connect the algebraic type to the existing implementation.

Listing 1 illustrates a simple example of a Tom program. The program starts with a Java class definition. The `%gom` construct defines an algebraic data type with one module, `Peano`. The module defines a sort `Nat` with two constructors: `zero` and `suc`. The constructor `suc` takes a variable `n` as a field. Given a signature, a well-formed and well-typed term can be built using the backquote (`‘`) construct. For example, for `Peano`, `‘zero()` and `‘suc(zero())` are correct terms, while `‘suc(zero(),zero())` or `‘suc(3)` are not well formed and not well typed, respectively.

In the example in List. 1, the `plus()` and `greaterThan()` methods are implemented by pattern matching. The semantics of pattern matching in Tom is close to the

[3] `http://www.eclipse.org/modeling/emf/`

```
public class PeanoExample {

 %gom {
  module Peano
  Nat = zero() | suc(n: Nat)
 }

 public Nat plus(Nat t1, Nat t2) {
  %match(t1,t2) {
   x,zero() -> { return `x; }
   x,suc(y) -> { return `suc(plus(x,y)); }
  }
```

```
boolean greaterThan(Nat t1, Nat t2) {
 %match(t1,t2) {
   x,x         -> { return false; }
   suc(_),zero() -> { return true; }
   zero(),suc(_) -> { return false; }
   suc(x),suc(y) -> { return `greaterThan(x,y); }
  }
}

public final static void main(String[] args) {
  Nat N = `zero();
  for(int i=0 ; i<10 ; i++) { N = `suc(N); }
} }
```

List. 1: An example of a Java and Tom program. Tom parts are in bold.

match which exists in functional programming languages, but in an imperative context. A `%match` construct is parametrised by a list of subjects, *i.e.*, expressions evaluated to ground terms, and contains a list of rules. The left-hand side of the rules are patterns built upon constructors and new variables, without any restriction on linearity (a variable may appear twice, as in `x,x`). The right-hand side is a Java statement that is executed when the pattern matches the subject. Using the backquote construct (`) a term can be created and returned. Similar to the standard `switch/case` construct, patterns are evaluated from top to bottom. In contrast to the functional *match*, several actions, *i.e.*, right-hand side, may be fired for a given subject as long as no `return` or `break` is executed.

In addition to the syntactic matching capabilities illustrated above, Tom also supports more complex matching theories such as matching modulo associativity, associativity with neutral element, and associativity-commutativity.

2.2 Generalised LL Parsing

Top down parsers whose execution closely follows the structure of the underlying grammar are attractive, particularly because they make grammar debugging easier. GLL is a top down parsing technique which is fully general, allowing even left recursive grammars, which has worst-case cubic runtime order and which is close to linear on most 'real' grammars. In this section we give a basic description of the technique, a full formal description can be found in [7].

A GLL parser effectively traverses the grammar using the input string to guide the traversal. There may be several traversal threads, each of which has a pointer into the grammar and a pointer into the input string. For each nonterminal A there is a block of code corresponding to each alternate of A. At each step of the traversal, (i) if the grammar pointer is immediately before a terminal we attempt to match it to the current input symbol; (ii) if it is immediately before a nonterminal B then the pointer moves to the start of the block of code

associated with B; (iii) if it is at the end of an alternate of A then it moves to the position immediately after the instance of A from which it came. This control flow is essentially the same as for a classical recursive descent parser in which the blocks of code for a nonterminal X are collected into a *parse function* for X with traversal steps of type (ii) implemented as a function call to X and traversal steps of type (iii) implemented as function return. In classical recursive descent, we use the runtime stack to manage actions (ii) and (iii) but in a general parser there may be multiple parallel traversals arising from nondeterminism; thus in the GLL algorithm the call stack is handled directly using a Tomita-style graph structured stack (GSS) which allows the potentially infinitely many stacks arising from multiple traversals to be merged and handled efficiently.

Multiple traversal threads are handled using process descriptors, making the algorithm parallel rather than backtracking in nature. Each time a traversal bifurcates, the current grammar and input pointers, together with the top of the current stack and associated portion of the derivation tree, are recorded in a descriptor. The outer loop of a GLL algorithm removes a descriptor from the set of pending descriptors and continues the traversal thread from the point at which the descriptor was created. When the set of pending descriptors is empty all possible traversals will have been explored and all derivations (if there are any) will have been found. Details of the creation and processing of the descriptors by a GLL algorithm can be found in [7] and a more implementation oriented discussion can be found in [11].

The output of a GLL parser is a representation of all the possible derivations of the input in the form of a *shared packed parse forest* (SPPF), a union of all the derivation trees of the input in which nodes with the same tree below them are shared and nodes which correspond to different derivations of the same substring from the same nonterminal are combined by creating a packed node for each family of children. To make sure that descriptors contain only one tree node, the root of the current subtree, the SPPFs are binarised in the natural left associative manner, with the two left-most children being grouped under an intermediate node, which is in turn then grouped with the next child to the left, etc. It is this binarisation that keeps both the size of the SPPF and the number of descriptors worst-case cubic.

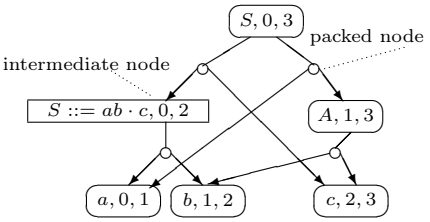

Fig. 1: SPPF

In detail, a binarised SPPF has three types of SPPF nodes: symbol nodes, with labels of the form (x, j, i) where x is a terminal, nonterminal or ϵ and $0 \leq j \leq i \leq n$; intermediate nodes, with labels of the form (t, j, i); and packed nodes, with labels for the form (t, k), where $0 \leq k \leq n$ and t is a grammar slot. (A grammar slot is essentially an LR(0)-item, a formal definition can be found

in [7].) Terminal symbol nodes have no children. Nonterminal symbol nodes, (A, j, i), have packed node children of the form $(A ::= \gamma\cdot, k)$ and intermediate nodes, (t, j, i), have packed node children with labels of the form (t, k), where $j \leq k \leq i$. A packed node has one or two children, the right child is a symbol node (x, k, i), and the left child, if it exists, is a symbol or intermediate node, (s, j, k). For example, for the grammar $S ::= abc \mid aA, \quad A ::= bc$ we obtain the SPPF as shown in Fig. 1. As is clear from this example, for ambiguous grammars there will be more than one derivation.

2.3 GLL Implementation Notes

Currently, the GLL parsing algorithm does not natively support EBNF; therefore, we convert a grammar described in EBNF to an equivalent BNF grammar prior to the generation of a GLL parser. In our conversion scheme, for example, an EBNF symbol $X*$, the repetition of X, is replaced by a nonterminal named X_*, having two alternates: $X\_* ::= X\ X\_*$ and $X\_* ::= \epsilon$. Because of these conversions, the resulting SPPF contains symbol nodes associated with these intermediary EBNF symbols. After the parsing, when the SPPF is created, the intermediary EBNF nodes are removed, by replacing them with their children, so that the resulting SPPF corresponds to the EBNF grammar.

Our GLL implementation uses a separate lexer. The lexer is driven by the parser and returns all possible token types seen at a particular point of the input. These tokens may overlap. Being a top-down parser, GLL decides at each grammar position whether tokens received from the lexer are relevant. This check is performed by testing the token types against the *first* and *follow* sets of nonterminals at the position. All the relevant tokens at a position are consumed and irrelevant ones are simply ignored.

The syntax of lexical definition used by our lexer is inspired by SDF [12]. In this syntax, a regular expression is defined as follows. A character `c` is represented by itself. `"."` is a special character matching all other characters. Meta characters, which have a specific meaning in the lexical definition, have to be escaped with a backslash to match the literal character they represent: `\\`, `\.`, `\^`, `\[`, `\]`, and `\-`. A range is represented by `a-b` where `a` and `b` are characters. A character class is defined as [$\texttt{r}_1\texttt{r}_2...\texttt{r}_n$] where each $\texttt{r}_i$ is either a character or a character range. [`^`$\texttt{r}_1\texttt{r}_2...\texttt{r}_n$] defines the negation of a character class. In our syntax, a character class is the smallest building block of a lexical definition, thus all characters should be defined inside a character class. If α and β are regular expressions, $\alpha\beta$, $\alpha|\beta$ are also regular expressions, denoting the concatenation of and the alternation between α and β, respectively. Moreover, $\alpha*$, $\alpha+$, and $\alpha?$ denote zero or more, one or more, and zero or one occurrence of α, respectively. Finally, A group can be specified by enclosing a regular expression within parentheses, *e.g.*, (α).

3 Island Grammars: Tom Syntax as an Example

Island grammars are a method for describing the syntax of a language, concentrating only on relevant constructs. An island grammar comprises two sets of rules: rules for *islands* describing the relevant language constructs which should be fully parsed and rules for *water* describing the rest of the text with which we are not concerned. In parsing embedded languages, the embedded language and host language constructs are captured as island and water, respectively. In this section, we define an island grammar for Tom. The presented approach is general enough to be used in other contexts as well.

```
context-free syntax
Program ::= Chunk*
Chunk ::= Water | Island
```

List. 2: The starting rules of an island grammar

The starting rules of an island grammar are shown in List. 2. As can be seen, a program is defined as a list of `Chunk` nonterminals, each being either an island or water. The rules defining water are presented in List. 3. In island grammars, the overlap between water and island tokens should be recognised. To this end, we define fine-grained `Water` tokens being as small as the building blocks of the supported host languages. In addition, our island grammar is designed to be host-language agnostic, although it may not possible to completely achieve it. For this purpose, the water definition should be flexible and capture the different varieties of tokens appearing in different supported host languages. For example, `SpecialChar` in List. 3 contains the union of all different symbols appearing in the host languages supported by our island grammar.

```
context-free syntax
Water ::= Identifier | Integer | String | Character | SpecialChar
lexical syntax
Identifier ::= [a-zA-Z_]+[a-zA-Z0-9\-_]*
Integer ::= [0-9]+
String ::= ["]([^"\\]|[\\]["trnu\\])*["]
Character ::= ['].['] | ['][\\][btnfr"'\\]['] 
SpecialChar ::= [; : + \- = & < > * ! % : ? | & @ \[ \] \^ # $ { } , \. ( )]
```

List. 3: The definition of `Water`

```
context-free syntax
Island ::= TomConstruct | BackQuoteTerm
TomConstruct ::= IncludeConstruct | MatchConstruct | GomConstruct | ...
```

List. 4: Tom constructs

As can be seen in List. 4, Tom islands fall into two groups: Tom constructs, prefixed by `%`, and the backquote term. Most of the Tom constructs have the same structure. In this section, we focus on two constructs: the `%include` construct,

which is the simplest construct of Tom, and the `%match` construct, which is one of the most used features of Tom. The syntax of these constructs is presented in List. 5. As can be seen, the production rule of `PatternAction` contains `Chunk*`, meaning that Tom and host-language constructs can be recursively nested inside a `%match` construct.

```
context-free syntax
IncludeConstruct ::= "%include" "{" Water* "}"

MatchConstruct ::= "%match" ( "(" MatchArguments ")" )? "{" PatternAction* "}"
MatchArguments ::= Type? Term ("," Type? Term)*
PatternAction ::= PatternList "->" "{" Chunk* "}"
Term ::= Identifier | VariableStar | Identifier "(" ( Term ("," Term)* )? ")"
lexical syntax
VariableStar ::= [a-zA-Z_]+[a-zA-Z0-9\-_]* [*]
Type ::= [a-zA-Z_]+[a-zA-Z0-9\-_]*
```

List. 5: The `%include` and `%match` constructs

Listing 6 introduces the backquote term. As can be seen, a new water type, `BackQuoteWater`, is introduced in the backquote term definition. The difference between `BackQuoteWater` and `Water` is that the former does not contain parentheses, dot, or commas, while the latter does. These characters should be captured as part of a backquote term's structure and not as water. Examples of backquote terms are `` `x ``, `` `x* ``, `` `f(1 + x) ``, `` `f(1 + h(t), x) ``, and `` `(f(x)%b) ``. The idea behind backquote terms is to combine classical algebraic notations of term rewriting with host language code such as `"1 +"` or `"%b"`. Note that `x` and `f(x)` inside a backquote term can be interpreted as Tom terms or Java variables and method calls. Based on the defined grammar in List. 6 and disambiguation rules in Sect. 4.3, `x` and `f(x)` are recognized as Tom terms. In the compilation phase, if these terms were not valid Tom terms, they will be printed to the output as they are, thus producing Java variables and method calls.

```
context-free syntax
BackQuoteTerm ::= "`" CompositeTerm | "`" "(" CompositeTerm+ ")"
CompositeTerm ::= VariableStar | Variable
            | Identifier "(" (CompositeTerm+ ("," CompositeTerm+)*)? ")"
            | BackQuoteWater
BackQuoteWater ::= Identifier | Integer | String | Character | BackQuoteSpecialChar
lexical syntax
BackQuoteSpecialChar ::= [; : + \- = & < > * ! % : ? | & @ \[ \] \^ # $ { }]
```

List. 6: The backquote term

4 Disambiguation

In this section we propose a pattern matching technique to resolve the ambiguities present in island grammars. The pattern matching is performed after parsing, when the SPPF is fully built. As discussed in Sect. 2.2, an SPPF provides efficient means for representing ambiguities by sharing common subtrees.

An ambiguity node in an SPPF is a symbol or an intermediate node having more than one packed node as children. For example, consider the grammar of arithmetic expressions in List. 7. For this grammar, the input string `"1+2+3"` is ambiguous. Its corresponding SPPF is presented in Fig. 2, which contains an ambiguity occurring under the root symbol node.

```
context-free syntax
E ::= E "+" E | Digit
lexical syntax
Digit ::= [1-9]+
```

List. 7: A simple grammar for arithmetic expressions

For SPPF visualisation, packed nodes are represented by small circles, intermediate nodes by rectangles, and symbol nodes by rounded rectangles in which the name of the symbol node is written. If a symbol node represents a terminal, the matched lexeme is also shown next to the terminal name, *e.g.*, `Digit: 1`. Keywords, *e.g.*, `"*"`, are represented by themselves without the quotation marks. Furthermore, for a more compact visualisation, nodes related to the recognition of layout, whitespace and comments, are excluded from the figures in this section. In our GLL implementation, layout is recognised after each terminal and before the start symbol of a grammar. Layout is captured through the nonterminal `Layout`, which is defined as `Layout ::= LayoutDef*`, where `LayoutDef` is a lexical definition of a layout, *e.g.*, whitespace.

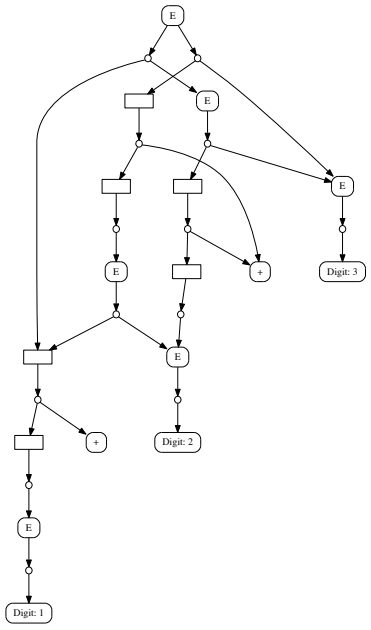

Fig. 2: The complete SPPF

While an SPPF is built, intermediate nodes are added for binarisation, which is essential for controlling the complexity of the GLL parsing algorithm. Furthermore, the version of the GLL algorithm we are using creates packed nodes even when there are no ambiguities. These additional nodes lead to a derivation structure which does not directly correspond to the grammar from which the SPPF has been created.

After the successful construction of an SPPF, one can traverse the SPPF and remove all packed nodes which are not part of an ambiguity, *i.e.*, the packed nodes which are the only child. The intermediate nodes which only have one child, the ones which do not present an ambiguity, can also be removed. When a node is removed, its children are attached to the parent of the node. Care should be taken that the order of children in the parent remains intact. By removing

unnecessary packed and intermediate nodes, the SPPF presented in Fig. 2 can be reduced to the SPPF in Fig. 3.

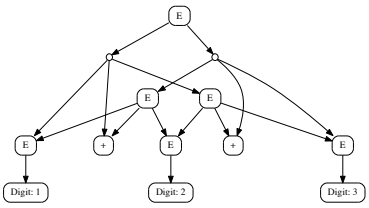

Fig. 3: The reduced SPPF

To resolve ambiguities in an SPPF, we traverse a reduced SPPF to find the deepest ambiguity node. At the ambiguity node, its children are checked against a set of disambiguation rules. A disambiguation rule is a rewrite rule that can either (i) remove a packed node matching an "illegal" pattern, or (ii) prefer a packed node over another one. These rules are called *remove* and *prefer* rules, respectively. In *prefer* rules, none of the patterns is illegal, but if both patterns appear under an ambiguity node, one of them should be preferred. A disambiguation rule may apply to more than one packed node under an ambiguity node or may not apply at all.

After applying the set of disambiguation rules on an ambiguity node, if only one packed node remains, the disambiguation is successful. Then, the remaining packed node is replaced by its children. This resolves the ambiguity. To completely disambiguate an SPPF, disambiguation is performed bottom up. This is because, in many cases, resolving an ambiguity in higher level ambiguity nodes in an SPPF depends on first resolving the ambiguities in deeper levels, as subtrees of an ambiguity node in a higher level may be ambiguous themselves. By performing the disambiguation bottom up, it is ensured that the subtrees of an ambiguity node are not ambiguous, thus they can uniquely be specified by tree patterns. If the entire disambiguation procedure is successful, an SPPF is transformed into a parse tree only containing symbol nodes.

The syntax for writing disambiguation rules as follows:

- A *remove* rule is written as `remove P`, where `P` is a pattern matching a packed node under an ambiguity node.
- A *prefer* rule is written as $\texttt{prefer P}_\texttt{i}\ \texttt{P}_\texttt{j}$, where $\texttt{P}_\texttt{i}$ and $\texttt{P}_\texttt{j}$ are patterns matching sibling packed nodes under an ambiguous node, in which $\texttt{P}_\texttt{i}$ should be preferred to $\texttt{P}_\texttt{j}$. Note that in this notation the list of packed nodes is considered modulo associativity-commutativity (AC), *i.e.*, as a multiset data-structure, and thus the order in which packed nodes appear does not matter.
- A packed node is written as $[\texttt{t}_1, \texttt{t}_2, ..., \texttt{t}_\texttt{n}]$, where each $\texttt{t}_\texttt{i}$ is a pattern describing a symbol node.
- A symbol node whose children are not important for us is represented by its label, *e.g.*, `E`. If the symbol node represents a keyword, it should be surrounded by double-quotes, *e.g.*, `"+"`.
- A symbol node in the general form is written as $\texttt{l}(\texttt{t}_1, \texttt{t}_2, ..., \texttt{t}_\texttt{n})$, where `l` is the label of the symbol node, and each $\texttt{t}_\texttt{i}$ is a pattern describing a child symbol node. Note that after removing the unnecessary packed and intermediate nodes, packed nodes can only appear as direct children of an ambiguity

node. Therefore, in defining the symbol nodes, no packed node can appear as their children. Using this syntax, the tree under a packed node can be described in as much detail as needed by expanding symbol nodes.
- `"_"` and `"_*"` match any symbol node and zero or more occurrence of any symbol nodes, respectively.

Note that writing patterns using this syntax does not require the user to explicitly specify layout nonterminals. Layouts are automatically captured after each terminal symbol, when the patterns are translated to Tom patterns, see Sect. 4.4. The ambiguity shown in Fig. 3 can be resolved by selecting the left-associative derivation, *i.e.*, `(1+2)+3`, rather than the other derivation, `1+(2+3)`. For this purpose, we define the right-associative derivation as illegal and remove it using the following rule: `remove [E, "+", E(E, "+", E)]`. As can be seen, the disambiguation rule closely follows the grammar rules and the resulting SPPF.

4.1 The Island-Water Ambiguity

Tom islands, with the exception of the backquote term, do not have unique opening and closing tags. We define a token as unique if it only appears in either the host or the embedded language. The lack of unique opening and closing tags may mislead the parser into recognising a Tom construct as a collection of successive water tokens. This leads to an ambiguity which we call the island-water ambiguity.

Consider the starting rules of Tom's island grammar in List. 2. When a GLL parser is about to select the alternates of `Chunk`, if the current input index points to a percentage sign followed by a Tom construct name, such as `include` or `match`, both alternates of `Chunk` are selected. Processing both alternates may lead to an island-water ambiguity. The SPPF corresponding to parsing `"%include {file}"` is illustrated in Fig. 4.

The island-water ambiguity can be resolved by preferring an island to a sequence of water tokens. This can be achieved by the following rewrite rule:

```
prefer [Chunk(Island)][Chunk(Water), _*]
```

As for this example, if the ambiguity contains more than one water token, the ambiguity occurs under a node labelled `Chunk_*`, which is in fact an intermediary node resulting from the EBNF to BNF conversion, see Sect. 2.3. The conversion, however, does not affect the writing of patterns, as patterns describe tree structures under packed nodes and not their parent, the ambiguous node.

4.2 The Island-Island Ambiguity

Detecting the end of a Tom construct is significantly more difficult than its beginning. Tom constructs end with a closing brace, which is also the end-of-block indicator in the supported host languages of Tom. Detecting the end of

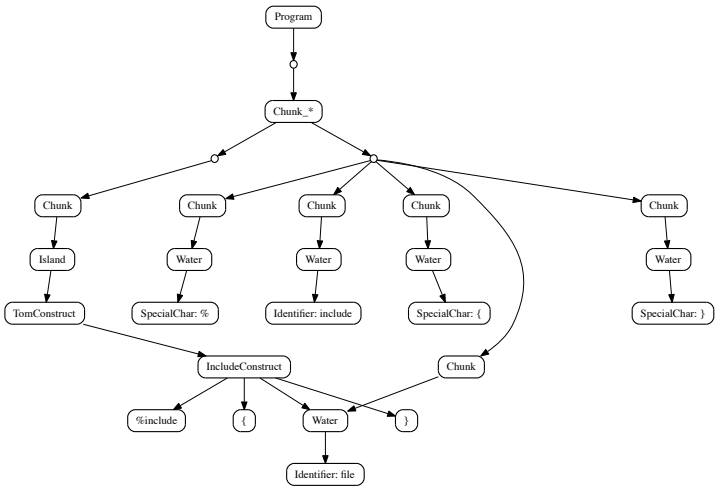

Fig. 4: The Island-Water ambiguity in an `%include` construct

a Tom construct may lead to what we call the island-island ambiguity, which happens between islands with different lengths. In this kind of ambiguity, some closing braces have been wrongly interpreted as water.

```
1 %match(s) {
2   x -> {
3     { System.out.println(‘x); }
4   }
5 }
```

List. 8: A `%match` construct containing an Island-Island ambiguity

An example of a Tom program containing an island-island ambiguity is given in List. 8. Parsing this example produces two different match constructs, and hence an ambiguity. A match construct needs at least two opening and closing braces to be successfully recognised, and the rest of the tokens, including any other closing braces, may be treated as water. For this example, every closing brace, after the second closing brace on line 4, may be interpreted as the closing brace of the match construct.

Resolving the island-island ambiguity is difficult, mainly because it depends on the semantics of the embedded and host languages. We disambiguate this case by choosing the match construct which has well-balanced braces in its inner water fragments. There are two ways to check the well-balancedness of braces. First, we can use a manual traversal function which counts the opening and closing braces inside water fragments of a Tom construct. This check is, however, expensive for large islands. Second, we can prevent islands with unbalanced braces to be created in the first place by modifying the lexical definition of `SpecialChar` in List. 3. We remove the curly braces from the lexical definition and add `"{" Chunk* "}"` as a new alternate to `Water`. With this modification, opening

and closing braces can only be recognised as part of a match construct or pairs of braces surrounding water tokens. We chose the second option for resolving the island-island ambiguities occurring in Tom's island grammar.

4.3 The Backquote Term Ambiguities

Identifying the beginning of a backquote term is straightforward because the backquote character does not exist in Tom's supported host languages. Therefore, no ambiguity occurs at the beginning of a backquote term. However, a number of ambiguities may occur while detecting the end of a backquote term. The simplest example of an ambiguous backquote term is `‘x`, in which `x` can either be recognised as `BackQuoteWater` or `Variable`. This ambiguity is depicted in Fig. 5. For disambiguation, we prefer a `Variable` over `BackQuoteWater` by the following disambiguation rule:

```
prefer [Variable][BackQuoteWater]
```

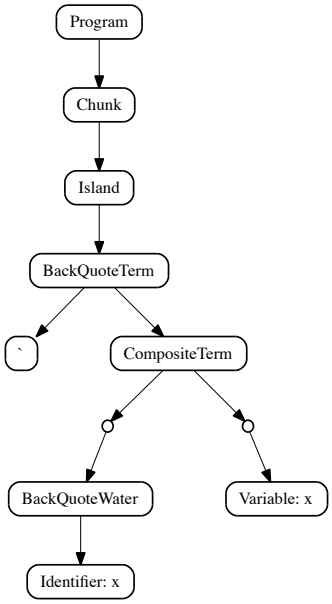

Fig. 5: `BackQuoteWater/Variable`

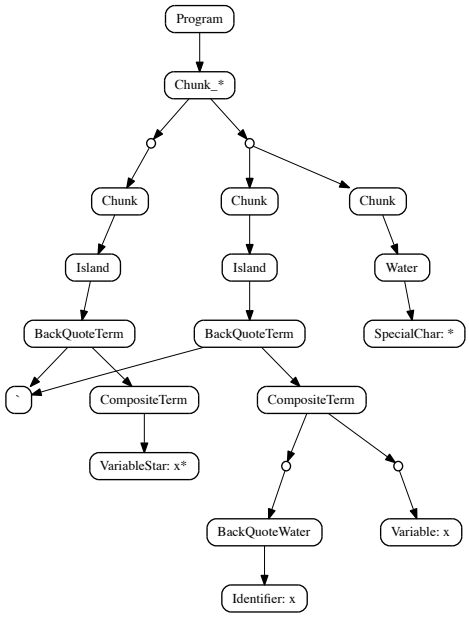

Fig. 6: `Variable/VariableStar`

Another example of ambiguity in the backquote term can be observed in `‘x*`, which can be recognised as a backquote term containing `VariableStar`, denoting a list of terms, or a backquote term containing the variable `x` after which a water token (the asterisk character) follows. The disambiguation in this case depends on the typing information of the host language. For example, `‘x* y`, considering

Java as the host language, should be recognised as `‘x` multiplied by `y`, and not `‘x*` followed by `y`, provided that `y` is an integer. We do not, however, deal with the typing complexities and currently follow a simple rule: if an asterisk character directly follows a variable, the recognition of `VariableStar` should be preferred. The SPPF corresponding to the parsing of `‘x*` is shown in Fig. 6.

In the SPPF in Fig. 6 two ambiguities are present: one between the recognition of `x` as `BackQuoteWater` or `Variable`, which is already discussed. The new ambiguity, caused by the presence of the asterisk character is under `Chunk_*`, can be resolved by the following rule:

```
prefer [Chunk(Island(BackQuoteTerm))][Chunk(Island(BackQuoteTerm)),Chunk(Water)]
```

An ambiguity similar to the one present in `‘x*` occurs in backquote terms ending in parentheses, see List. 6. For example, in `‘x()`, parentheses can be recognised as water and not as part of the backquote term. We use a disambiguation rule in which a backquote term containing the parentheses as part of the backquote term should be preferred to the one in which parentheses are recognised as water.

4.4 Implementation

So far, we described our disambiguation mechanism without exploring the implementation details. To implement disambiguation rules, we use the pattern matching and rewriting facilities of Tom. Listing 9 defines an algebraic data type for an SPPF using a `%gom` construct. The algebraic type defines a sort `SPPFNode` with three constructors, corresponding to three node types present in an SPPF, and the sort `NodeList` defining a list of nodes. The `SymbolNode` constructor creates a symbol node with its label and its list of the children. The `PackeNode` and `IntermediateNode` constructors create a packed or intermediate node by their list of children. Finally, `concNode` is the constructor for creating a list of `SPPNode` terms using the $*$ operator.

```
%gom {
  SPPFNode = SymbolNode(label:String, children:NodeList)
         | PackedNode(children:NodeList)
         | IntermediateNode(children:NodeList)
  NodeList = concNode(SPPFNode*)
}
```

List. 9: The algebraic type definition for an SPPF

In addition to the algebraic signature in List. 9, we use a Tom *mapping* which connects the Java implementation of an SPPF, used by the parser, to the algebraic view. For example, using the mapping, an instance of the `SymbolNode` class, from the Java implementation, is viewed as a `SymbolNode` algebraic constructor. The subterms of this constructor, which are of sort `String` and `NodeList`, are then automatically retrieved by the mapping. More importantly, through this algebraic view, the Java implementation can be automatically traversed using Tom

strategies, without the need to provide hand-written visitors. All this efficient and statically-typed machinery is generated and optimised by the Tom compiler.

```
SymbolNode("E", concNode(z1*,
  PackedNode(concNode(
      SymbolNode("E",_)
      SymbolNode("+",_),_,
      SymbolNode("E", concNode(SymbolNode("E",_), SymbolNode("+",_), _, SymbolNode("E",_)))
  )),z2*)) -> SymbolNode("E", concNode(z1*, z2*));
```

List. 10: A rewrite rule in Tom for disambiguating binary operators

The rule for removing the right-associativity, *i.e.*, `remove [E, "+", E(E, "+", E)]`, is translated to the Tom rewrite rule in List. 10. The notation `concNode(z1*, PackedNode(...), z2*)` denotes *associative* matching with *neutral element*, meaning that the subterm `PackedNode` is searched into the list, the *context*, possibly empty, being captured by variables `z1*` and `z2*`. Furthermore, as layout tokens may be present after each terminal symbol, the anonymous variable `"_"` is automatically added after each terminal to capture the layout.

5 Results

Using the island grammar and the disambiguation rules presented in Sects. 3 and 4, respectively, we were able to parse all the Tom programs from the tom-examples package. This package, which is shipped with the source distribution of Tom, contains more than 400 examples (for a total of 70,000 lines of code) showing how Tom is used in practice. The size of these examples varies from 24 lines of code (679 characters) to 1,103 lines of code (30,453 characters).

Based on our findings, the examples having a size of about 10,000 characters could be parsed and disambiguated in less than one second, and the disambiguation time of an instance was always lower than its parsing time. Moreover, from what we observed, the parsing time for Tom examples was linear.

Table 1: Parsing times for the selected Tom examples (in milliseconds)

Example file	#Lines	#Tokens	Parsing time	Disambiguation time	Total time
Peano.t	84	340	362	64	426
Compiler.t	169	1,365	718	304	1,022
Analysis.t	253	1,519	667	358	1,025
Langton.t	490	3,430	1,169	524	1,693
TypeInference.t	909	6,503	1,632	850	2,482
BigExample.t	1,378	11,682	2,917	739	3,656

In order to evaluate the efficiency of our approach, we selected five representative examples, see Table 1. Peano is a simple example manipulating Peano integers. Compiler is a compiler for a Lazy-ML implementation. Analysis is a bytecode

static analyser based on CTL formulae. Langton is a cellular automata simulator, which contains many algebraic patterns. TypeInference is a type-inference algorithm for patterns, in presence of sub-typing. This example contains many nested `%match` constructs. We also considered an artificial example, BigExample, that could be made as big as needed, by concatenating pieces of code.

We compared our implementation with the current implementation of the Tom parser, which also uses an island-grammar approach, implemented using the ANTLR parser generator. Currently, the ANTLR 3-based implementation is approximatively twice as fast on a few examples than the GLL implementation. However, our GLL implementation is not yet thoroughly optimised. In particular, the scanner can be made more efficient. Therefore, we expect to obtain better results in the future.

6 Related work

The name "island grammar" was coined in [13] and later elaborated on in [3]. Moonen showed in [3] how to utilise island grammars, written in SDF, to generate robust parsers for reverse engineering purposes. Robust parsers should not fail when parsing incomplete code, syntax errors, or embedded languages. However, the focus of [3] is more on reverse engineering and information extraction from source code rather than parsing embedded languages.

There has been considerable effort on embedding domain-specific languages in general-purpose programming languages. For example, Bravenboer and Visser [14] presented MetaBorg, a method for providing concrete syntax for object-oriented libraries, such as Swing and XML, in Java. The authors demonstrated how to express the concrete syntax of language compositions in SDF [12], and transform the parsed embedded constructs to Java by applying rewriting rules using Stratego[4]. In MetaBorg the full grammar of Java is parsed, while we use an island-grammar based approach. Furthermore, MetaBorg uses distinct opening and closing tags for transition between the host and embedded languages which prevents many ambiguities to happen in the first place.

The ASF+SDF Meta-Environment[5] has been used in a number of cases for developing island grammars. SDF, the underlying syntax formalism of the ASF+SDF Meta-Enviornment, provides the *prefer* and *avoid* mechanisms to mark nodes in the parse forest which should be kept or removed [15]. These mechanisms, however, do not work in all cases and may produce unpredictable results. For example, [16,17] show examples of island grammars which cannot be disambiguated using SDF. The problem with *prefer* and *avoid* mechanisms is that the decision for selecting the desired parse tree under an ambiguity node is made by only considering the highest level production rule, which are marked with *prefer*

[4] `http://strategoxt.org/`

[5] `http://www.meta-environment.org/Meta-Environment/ASF+SDF`

or *avoid*. In many cases, the disambiguation cannot be done by merely checking the highest level rule, and there is a need to explore the subtrees in more detail. Our disambiguation syntax allows the user to express the trees under ambiguity nodes using patterns in as much detail as needed.

An example of using island grammars in parsing embedded languages which does not use a generalised parsing technique is presented by Synytskyy et al. in [18]. The authors presented an approach for robust multilingual parsing using island grammars. They demonstrated how to parse Microsoft's Active Server Pages (ASP) documents, a mix of HTML, Javascript, and VBScript, using island grammars. For parsing, they used TXL[6], which is a hybrid of functional and rule-based programming styles. TXL uses top-down backtracking to recognise the input. Due to too much backtracking, for some grammars, the parsing might be very slow or impractical [19]. Moreover, a grammar expressed in TXL is disambiguated by ordering the alternates. In contrast to this work, we used a generalised parser which has the worst case complexity of $O(n^3)$. Moreover, our disambiguation mechanism covers a wider range of ambiguities than alternate ordering.

Another related work which does not use a generalised parsing technique is presented by Schwerdfeger and Van Wyk in [20]. The authors describe a mechanism for verifying the composability of deterministic grammars. In their approach, grammar extensions are checked for composability, and if all extensions pass the check, an LR parser and a context-aware scanner is generated for the composition. The context-aware scanner is used to detect the overlaps between tokens. The presented approach in [20] parses the full grammar of the host language instead of an island grammar. Furthermore, as mentioned by the authors, not all extensions pass the composiblitiy check. Our approach is more generic compared to [20] since by using GLL no restriction exists on composing grammars. More importantly, as explained in Sect. 2.3, a GLL parser with a separate lexer only consumes token types which are relevant at the parsing position, thus effectively providing context-sensitive lexing, without the need to modify the parser or change the parsing algorithm. Finally, using GLL and our disambiguation mechanism, we are able to deal with complex, ambiguous grammars.

For island grammar-based parsing, it is essential to deal with tokens which overlap or have different types. Aycock and Nigel Horspool in [21] proposed the Schrödinger's token method, in which the lexer reports the type of a token with multiple interpretations as the Schrödinger type, which is in fact a meta type representing all the actual types. When the parser receives a Schrödinger's token, for each of its actual types, parsing will be continued in parallel.

An alternative to Schrödinger's token is to use scannerless parsing [6]. In scannerless parsing there is no explicit lexer present, so the parser directly works on the character level. This has the advantage of giving the parser the opportunity to decide on the type of a token based on the context in which the token's char-

[6] `http://www.txl.ca/`

acters appear. Scannerless parsing has mainly two disadvantages. First, because every character in the language is a token, the size of the parse forest is larger than parsing with a scanner. Second, ambiguities happening at the character level, *e.g.*, the longest match for keywords, should be explicitly resolved in the parse tree. A GLL parser with a separate scanner provides the same power as a scannerless parser, without having to deal with character level ambiguities, as they are already resolved by the scanner.

7 Conclusions and Future Work

In this paper we have shown how languages based on island grammars can be parsed using existing technologies such as GLL and Tom. We developed a host-agnostic island grammar for Tom which only fully parses the Tom constructs and ignores the host language constructs. The primary challenge of island grammar-based parsing is the ambiguity resulting from the overlap between host and embedded language tokens. We analysed different ambiguity classes of Tom's island grammar and provided patterns for resolving them.

Our main objective was to avoid modification within the parser in order to provide a generic solution for parsing embedded languages. We proposed a method for disambiguating island grammars using pattern matching and rewriting the parse forest. Our disambiguation mechanism is implemented by the pattern matching mechanism of Tom. In addition, in one case, the island-island ambiguity, we rewrote the grammar to resolve the ambiguity. Using our approach, one can express a modular island grammar in EBNF, independent of the complexity or the number of embedded languages.

We performed a number of experiments to validate our findings. In these experiments, we parsed all the Tom files from the tom-examples package, with different characteristics such as different sizes and number of islands. Moreover, we compared our GLL implementation with the current ANTLR implementation. The GLL implementation has a parsing speed comparable to the current implementation, but in a few cases the ANTLR implementation is twice as fast.

As future work, there are a number of paths we shall explore. First, we will investigate how we can improve our GLL implementation as it is not yet thoroughly optimised. Second, we shall work on replacing the current ANTLR implementation with the GLL implementation in the Tom compiler. Third, we will investigate the possibility of pattern matching while parsing to increase the efficiency of the disambiguation process. Finally, we plan to apply our approach to other language compositions, *e.g.*, Java+XML or Java+SQL, to determine how generic our method is.

Acknowledgements. We would like to thank Alexander Serebrenik who has provided us with feedback on an early draft of this work and also helped us in measuring the complexity of parsing Tom examples.

References

1. Moreau, P.E., Ringeissen, C., Vittek, M.: A pattern matching compiler for multiple target languages. In: Proceedings of the 12th international conference on Compiler construction, Springer-Verlag (2003) 61–76
2. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. Science of Computer Programming **75**(7) (2010) 473–495
3. Moonen, L.: Generating robust parsers using island grammars. In: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01). (2001) 13–
4. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Norwell, MA, USA (1985)
5. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam, The Netherlands (1992)
6. Visser, E.: Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam (1997)
7. Scott, E., Johnstone, A.: GLL parse-tree generation. Science of Computer Programming (2012) To appear
8. Manders, M.W.: mlBNF - a syntax formalism for domain specific languages. Master's thesis, Eindhoven University of Technology, The Netherlands (2011)
9. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking Rewriting on Java. In: Proceedings of the 18th international conference on Term rewriting and applications. RTA'07, Springer-Verlag (2007) 36–47
10. Balland, E., Kirchner, C., Moreau, P.E.: Formal islands. In: Proceedings of the 11th international conference on Algebraic Methodology and Software Technology. AMAST'06, Springer-Verlag (2006) 51–65
11. Johnstone, A., Scott, E.: Modelling GLL parser implementations. In: Proceedings of the Third international conference on Software language engineering. SLE'10, Springer-Verlag (2011) 42–61
12. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF-reference manual-. SIGPLAN Not. **24**(11) (1989) 43–75
13. van Deursen, A., Kuipers, T.: Building documentation generators. In: Proceedings of the IEEE International Conference on Software Maintenance. (1999) 40–
14. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. SIGPLAN Not. **39**(10) (2004) 365–383
15. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: Proceedings of the 11th International Conference on Compiler Construction, Springer-Verlag (2002) 143–158
16. Post, E.: Island grammars in ASF+SDF. Master's thesis, University of Amsterdam, The Netherlands (2007)
17. van der Leek, R.: Implementation Strategies for Island Grammars. Master's thesis, Delft University of Technology, The Netherlands (2005)
18. Synytskyy, N., Cordy, J.R., Dean, T.R.: Robust multilingual parsing using island grammars. In: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research. CASCON '03, IBM Press (2003) 266–278
19. Cordy, J.R.: TXL - A Language for Programming Language Tools and Applications. Electronic Notes in Theoretical Computer Science **110** (2004) 3–31
20. Schwerdfeger, A.C., Van Wyk, E.R.: Verifiable composition of deterministic grammars. SIGPLAN Not. **44**(6) (2009) 199–210
21. Aycock, J., Nigel Horspool, R.: Schrödinger's Token. Software, Practice & Experience **31** (2001) 803–814