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], Mark van den Brand[1], Adrian Johnstone[4], Maarten Manders[1], Pierre-Etienne Moreau[2,3], and Elizabeth Scott[4]

[1] Eindhoven University of Technology, NL-5612 AZ Eindhoven, The Netherlands
[2] Inria, Villers-lès-Nancy, 54600, France
[3] Université de Lorraine, LORIA, Vandœuvre-lès-Nancy, 54500, France
[4] Department of Computer Science, Royal Holloway, University of London, Egham, Surrey, United Kingdom

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 fragments are encoded, in which case the parser of the host language does not need to be aware of the fact that another language is present. The drawback is that syntax errors in the embedded language fragments are not detected until a later phase in which the embedded language fragments are processed.

In the last decade language developers have been working on extending general-purpose programming languages with domain-specific languages, referred to as guest languages. The language embeddings which are not simple string encodings present a challenge from a parsing point of view. Especiallly, if the general-purpose programming and the embeddings can be unboundedly nested. Tom [12] is an example of such an extension which provides a general-purpose programming language, 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.

The syntax of the host language needs to be modified to accommodate the guest language if the language embeddings are not simple string encodings. 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 [3]. 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 [11]. An island grammar captures the important constructs, embedded constructs in our case, as "island"s 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 might not 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 be in the same class. Second, the host and extension languages may share tokens, which may lead to ambiguities. The ambiguities from shared tokens cannot be always resolved by rewriting the grammar, using more lookahead 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 [18] there have been a number of generalised LR parsing (GLR) implementations, such as GLR [14], a scannerless variant (SGLR) [20] and Dparser[2]. Johnstone and Scott developed a generalised LL (GLL) parser [16] 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. We conducted the parsing experiments using an improved version of our Java implementation of GLL [10].

The rest of this paper is organised as follows. In Section 2 we introduce Tom as a language for term rewriting, the basics of context-free grammars and a brief description of the GLL parsing algorithm. Section 3 describes the idea of island grammars by defining an island grammar for Tom. In Section 4 we illustrate our mechanism for solving ambiguities in island-based grammars. We explain our disambiguation approach by providing rewrite rules for different types of ambiguities present in Tom. The results of parsing Tom examples are presented in Section 5. In Section 6 we present other work in the area of parsing embedded languages and compare our approach with them. Finally, in Section 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 embedded syntax with recursive nesting, which poses difficulties for conventional deterministic parsers. We define an island grammar for Tom and generate a GLL parser

[1] ANTLR home page. `http://www.antlr.org/`, Last visited: February 2012.
[2] DParser home page. `http://dparser.sourceforge.net/`, Last visited: June 2012.

for this grammar. Second, Tom is used as a tool for implementing a disambiguation mechanism for island grammars. Our disambiguation mechanism which is based on pattern matching and term rewriting is explained in Section 4. The rest of this section gives a brief introduction to context-free grammars and the GLL parsing algorithm. Furthermore, some notes about our GLL implementation are given.

2.1 Tom in a Nutshell

Tom [1,12] 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, Python or Caml. Tom relies on the Formal Island framework [2], 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 of 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 an 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 one can define an algebraic data type in Gom and 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() `` are `` `suc(zero()) `` are correct terms, while `` `suc(zero(),zero()) `` or `` `suc(3) `` are not well-formed and not well-typed, respectively.

[3] EMF home page. `http://www.eclipse.org/modeling/emf/`, Last visited: May 2012.

```
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); }
}
```

Listing 1: An example of a Tom program

In the example of Listing 1, the `plus()` and `greaterThan()` methods are implemented by pattern matching. The semantics of pattern matching in Tom is close to the *match* that exists in functional programming languages, but in an imperative context. A `%match` 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 Context-free Grammars

A *context-free grammar* (CFG) consists of a set $\mathbf{N}$ of nonterminals, a set $\mathbf{T}$ of terminals, a set of grammar rules of the form $A ::= \alpha$ where $A \in \mathbf{N}$ and α is a string in $(\mathbf{T} \cup \mathbf{N})^*$, and a specified start symbol, $S \in \mathbf{N}$. The symbol ϵ denotes the empty string. Rules with the same left hand side can be written as a single rule using the alternation symbol, $A ::= \alpha_1 \mid \ldots \mid \alpha_t$. The strings α_j are the *alternates of A*.

A grammar Γ defines a set of strings of terminals, its *language*, as follows. A *derivation step*, has the form $\gamma A \beta \Rightarrow \gamma\alpha\beta$ where $\gamma, \beta \in (\mathbf{T} \cup \mathbf{N})^*$ and $A ::= \alpha$ is a grammar rule. A *derivation* of τ from σ is a sequence $\sigma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$, also written $\sigma \overset{*}{\Rightarrow} \tau$. A derivation is *left-most* if at each step the left-most nonterminal is replaced. The language defined by Γ is the set of $u \in \mathbf{T}^*$ such that $S \overset{*}{\Rightarrow} u$. A grammar is *ambiguous* if there is a string u for which there is more than one left-most derivation $S \Rightarrow u$.

The role of a *parser* for Γ is, given a string $u \in \mathbf{T}^*$, to find (all) the derivations $S \stackrel{*}{\Rightarrow} u$. In our case the derivations will be recorded as trees. A *derivation tree* in Γ is an ordered tree whose root node is labelled with the start symbol, whose interior nodes are labelled with nonterminals and whose leaf nodes are labelled with elements of $\mathbf{T} \cup \{\epsilon\}$. The children of a node labelled A are ordered and labelled with the symbols, in order, from an alternate of A. The *yield* of the tree is the string of leaf node labels in left to right order. A grammar Γ is ambiguous if and only if there are two or more derivation trees in Γ with the same yield.

2.3 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 [16].

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 [16] and a more implementation oriented discussion can be found in [15].

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.

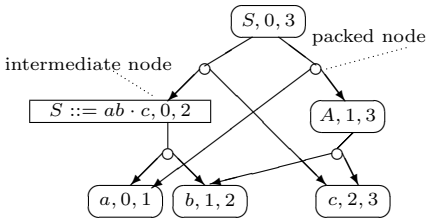

Figure 1: SPPF

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 [16].) 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 Figure 1. As is clear from this example, for ambiguous grammars there will be more than one derivation.

2.4 GLL Implementation Notes

The GLL parsing algorithm does not currently support EBNF; therefore, we convert EBNF to BNF before the generation of a GLL parser. In our conversion scheme, an EBNF symbol $X*$, *i.e.*, repetition of X, is rewritten to a nonterminal named X_*, which has the alternates $X\_* ::= X\ X\_*$ and $X\_* ::= \epsilon$. Because of these conversions, the generated SPPF may contain symbol nodes whose labels do not exit in the grammar. Therefore, it is essential to take the EBNF to BNF conversion into account while writing the disambiguation patterns.

Another point about our GLL implementation is that we use 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 the token type received from the lexer is relevant. This check is performed by testing the token types against the *first* and *follow* sets of alternates. All the relevant tokens at a position are consumed, and irrelevant ones are simply ignored.

The syntax of lexical definition used in our lexer is inspired by SDF [7] and Lex [9]. In this syntax, a regular expression is defined as follows. The character `c` is represented by `[c]` or `c`. `"."` is a special character which matches all other characters. Special characters should be escaped as: `\\`, `\.`, `\^`, `\[`, `\]`. 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. If α and β are regular expressions, $\alpha\beta$, $\alpha*$, $\alpha+$, and $\alpha?$ are regular expressions as well.

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 context-free rules. Rules for *islands* which describe the relevant language constructs which should be fully parsed, and rules for *water* which describe the rest of the text which we are not concerned with. In parsing an embedded language, islands are the embedded language constructs while water comprises the host language constructs. In this section, we define an island grammar for Tom. The presented approach is general enough to be used in other contexts as well. Listing 2 shows the starting rules of an island grammar.

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

Listing 2: The starting rules of an island grammar

The rules defining water are presented in Listing 3. We opted for water elements as small as the smallest building blocks of the host language, such as identifiers, keywords, and special characters. This allows the recognition of overlapping between the water and island tokens which is essential for detecting ambiguities.

```
context-free rules
Water ::= Identifier | Integer | String | Character | SpecialChar | "(" | ")" | ","
lexical rules
Identifier ::= [a-zA-Z_]+[a-zA-Z0-9\-_]*
Integer ::= [0-9]+
String ::= ["](~[^"\\]|[\\]["]|[\\][t]|[\\][r]|[\\][n]|[\\][u]|[\\][\\])*["]
Character ::= ['].['] | ['][\\] [btnfr"'\\] ['] 
SpecialChar ::= [; : + \- = & < > * ! % : ? | & @ \[ \] \^ # $ . { }]
```

Listing 3: The definition of `Water`

Our island grammar is designed to be as host-language agnostic as possible, although it may not possible to be completely host-language agnostic. Water definitions should be very flexible and capture the different varieties of tokens which may appear in different host languages which Tom supports. For example, `SpecialChar` contains the union of all different symbols which appear in the different host languages, so that we are able to capture all special characters with a single lexical definition.

Tom islands fall into two groups: constructs prefixed by `%`, described in Listings 4, 5, and 6, and backquote terms, illustrated in Listing 7, which allow nested and interleaved island and water constructs.

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

Listing 4: Tom constructs

Most of the Tom constructs have the same structure. In this section, we focus on two constructs: the `%include` construct (Listing 5) which is the simplest construct of Tom, and the `%match` construct (Listing 6) which is one of the most used features of Tom.

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

Listing 5: The `%include` construct

```
context-free rules
MatchConstruct ::= "%match" ( "(" MatchArguments ")" )? "{" PatternAction* "}"
MatchArguments ::= Type? Term ("," Type? Term)*
PatternAction ::= PatternList "->" "{" Chunk* "}"
Term ::= Identifier | VariableStar
       | Identifier "(" ( Term ("," Term)* )? ")"
lexical rules
VariableStar ::= Identifier "*"
Type ::= Identifier
```

Listing 6: The `%match` construct

As shown in Listing 6, the production rule of `PatternAction` contains `Chunk*`, which means that Tom and host-language constructs can be nested inside a match construct.

```
context-free rules
BackQuoteTerm ::= "`" CompositeTerm | "`" "(" CompositeTerm+ ")"
CompositeTerm ::= VariableStar | Variable
               | Identifier "(" (CompositeTerm+ ("," CompositeTerm+)*)? ")"
```

Listing 7: The backquote term

As can be seen from Listing 7, a new water type, `BackQuoteWater`, has been introduced in the backquote term definition. The difference between `BackQuoteWater` and `Water` is that the former does not contain parentheses or commas, while the

latter does. The parentheses and commas inside a backquote construct should be captured as part of the 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"`.

4 Disambiguation

In this section, we primarily focus on ambiguities resulting from shared tokens between the host and the guest language, which is the main ambiguity type in island grammars. Our assumption is that the guest language is disambiguated before embedding. We describe how ambiguities are represented in the SPPF, and how we can use the pattern matching facilities of Tom to disambiguate them. Our disambiguation is applied when the parsing has finished, and not during the SPPF construction. In the second part of this section, we illustrate different types of ambiguities present in the Tom's island grammar and provide patterns for disambiguating them.

4.1 Disambiguation by Pattern Matching within the SPPF

As discussed in Section 2.3, an SPPF provides efficient means for representing ambiguities by sharing the common subtrees. An ambiguity node in SPPF is a symbol or an intermediate node which has more than one packed node as children. For example, consider the grammar of arithmetic expressions defined in Listing 8.

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

Listing 8: A simple grammar for arithmetic expressions

For this grammar, the input string `"1+2+3"` is ambiguous, whose corresponding SPPF is presented in Figure 2, in which packed nodes are represented by small circles, intermediate nodes by boxes, and symbol nodes by rounded boxes in which the label of the symbol nodes are written. The SPPF contains an ambiguity which occurs below the first symbol node labelled "E".

While an SPPF is built, intermediate nodes are added for binarisation, which is essential for controlling the complexity of 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 expected abstract syntax of the grammar from which the SPPF has been created.

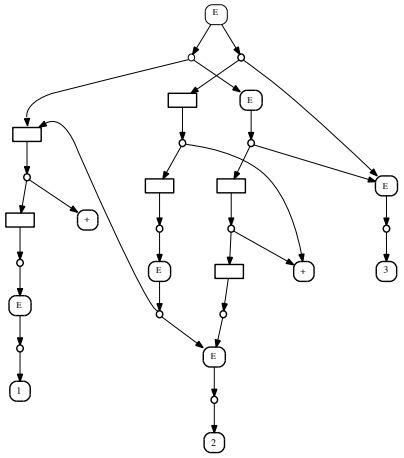

Figure 2: The raw SPPF for `1+2+3`

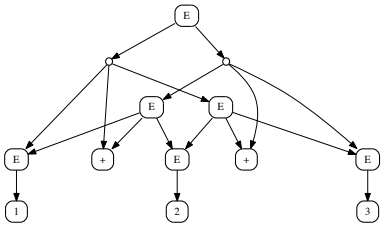

Figure 3: The reduced SPPF for `1+2+3`

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 Figure 2 can be reduced to the SPPF in Figure 3.

For disambiguating an SPPF, we use the pattern matching and rewriting facilities of Tom. Listing 9 defines an algebraic data type for 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`, which defines a list of children.

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

Listing 9: The algebraic type definition of an SPPF

In addition to the algebraic signature in Listing 9, we provide a *mapping* which connects the Java implementation of an SPPF, used by the parser, to the algebraic view. For example, an instance of `SymbolNode`, 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 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.

To detect and remove ambiguities, we traverse the SPPF to find an ambiguity node. At the ambiguity node, its children are checked against a set of disambiguation patterns. A pattern is a rewrite rule which prefers one packed node over another one, or discards packed nodes which contain an "unwanted" pattern. If the disambiguation is successful, only one packed node remains. It can be the case that a pattern cannot be applied on an ambiguity node or cannot completely disambiguate it. In this case, other patterns should be applied as well. If after applying all patterns still more than one packed node is present, the disambiguation fails. Otherwise, if the disambiguation is successful, the remaining packed node is replaced by its children. If the entire disambiguation procedure is successful, an SPPF is transformed into a parse tree which only contains symbol nodes.

The ambiguity shown in Figure 3 can easily be resolved by preferring the left-associative derivation, *i.e.*, the input `1+2+3` should be interpreted as `(1+2)+3` and not as `1+(2+3)`. Listing 10 shows how we disambiguate this example by a rewrite rule written in Tom.

```
SymbolNode("E", concNode(_*,
  PackedNode(x@concNode(
    SymbolNode("E", concNode(SymbolNode("E",_), SymbolNode("+",_), SymbolNode("E",_)))
    SymbolNode("+",_),
    SymbolNode("E",_)
  )),_*)) -> SymbolNode("E",x)
```

Listing 10: A rewrite rule in Tom for disambiguating binary operators

In Listing 10, the notation `x@t` means that the matched subterm `t` is stored in a variable named `x`, which can be reused in the right-hand side of the rule. The notation `concNode(_*,x@PackedNode(...),_*)` denotes *associative* matching with *neutral element*, meaning that the subterm `PackedNode` is searched into the list, the *context*, possibly empty, being captured by anonymous variables `_*`. The rewrite rule of Listing 10 has to be applied in a bottom-up way in order to remove deepest ambiguity nodes first. The process is then repeated if needed.

Expressing the patterns using the algebraic terms is tedious for users since it does not directly correspond to the parse tree. To overcome this, we present higher level syntax for writing disambiguation patterns. In our syntax the following simplifications are made:

- `SymbolNode(_,concNode(`$\mathtt{PN}_1, \mathtt{PN}_2, ..., \mathtt{PN}_n$`))`, which is an ambiguous symbol node whose packed node children are $\mathtt{PN}_1, \mathtt{PN}_2, ..., \mathtt{PN}_n$, is written as $\{\mathtt{PN}_1, \mathtt{PN}_2, ..., \mathtt{PN}_n\}$. Note that in this notation the list of packed nodes is considered modulo associativity-commutativity (AC), *i.e.*, as a multiset data-structure.
- `PackedNode(concNode(`$\mathtt{t}_1, \mathtt{t}_2, ..., \mathtt{t}_n$`))` whose children are $\mathtt{t}_1, \mathtt{t}_2, ..., \mathtt{t}_n$ is written as `PackedNode([`$\mathtt{t}_1, \mathtt{t}_2, ..., \mathtt{t}_n$`])`.
- `SymbolNode("E",_)`, a symbol node whose children are not important for us, is represented by `"E"`, its label.

– `SymbolNode("E",concNode(`$t_1, t_2, ..., t_n$`))` where $t_1, t_2, ..., t_n$ are the subterms of the `SymbolNode`, is written as `"E"(`$t_1, t_2, ..., t_n$`)`. In other words, the label of a symbol node becomes the root of the tree.
– `SymbolNode(`ϵ`)`, which represents the symbol node associated with the recognition of an empty string, is presented by `e`.

With this simplified notation, the rewrite rule of Listing 10 can be rewritten as: `{ PackedNode(x@["E"("E", "+", "E"),"+","E"]), y* } -> { x }`.

The rule is applied modulo AC on a `SymbolNode`. Modulo AC implies that `PackedNode` is searched into the multiset, and all remaining nodes are captured in a context variable `y*`. The subterms of the considered `PackedNode` are denoted by a variable `x`, which become the new list of children of the `SymbolNode`, on which the rule is applied (see Figure 3).

4.2 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 elements. This leads to an ambiguity which we call the island-water ambiguity.

Consider the starting rules of the Tom's island grammar from Listing 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 Figure 4. For a more compact visualisation we excluded the SPPF nodes related to the recognition of layout from the figures in this section.

The island-water ambiguity can be resolved by disallowing the presence of a water sequence containing `"%"` immediately followed by the name of a Tom construct as successive leaves in an SPPF. In contrast to the example of Listing 8, which only checks one packed node, we need to check two packed nodes to decide which one to keep. This is because the presence of a sequence of `"%"` and `"match"` is not always unwanted. This combination is only unwanted if another sibling packed node has an island beneath it. This means if the parser can recognise a part of the input as an island, it should be preferred over water. Preferring an island over water resembles the alternate orderings in PEGs [6]. The pattern for solving the island-water ambiguity is presented in Listing 11.

```
{ x@PackedNode(["Chunk"("Island"), "Chunk*"]),
  y@PackedNode(["Chunk"("Water"("%")), "Chunk*"("Chunk"("Water"), "Chunk*")]),
  z* } -> { x, z* }
```

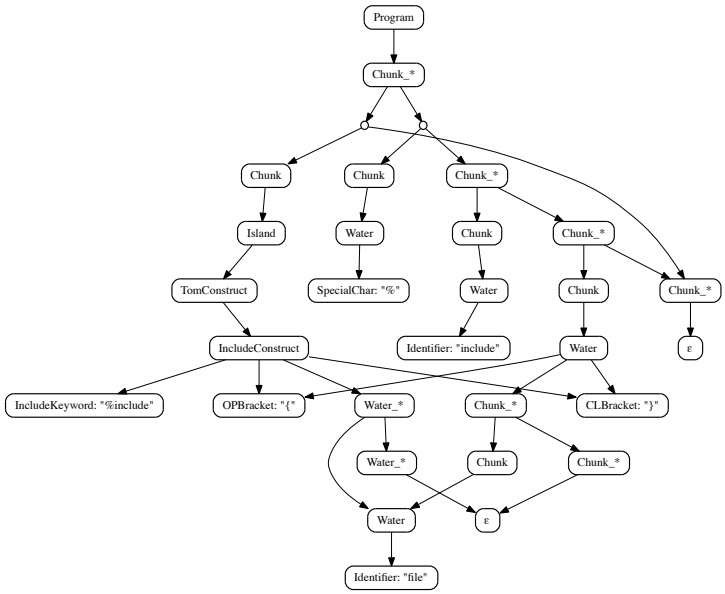

Figure 4: The Island-Water ambiguity for an `%include` construct

Listing 11: Disambiguation rule for the island-water ambiguity

The rule given Listing 11 also involves matching modulo AC. In this example, two `PackedNode` are searched, without any specific order, thanks to commutativity. The remaining nodes are captured by `z*`, which is put back in the right-hand side. Once more, the rule is applied in an innermost way.

4.3 The Island-Island Ambiguity

Detecting the end of a Tom construct is significantly more difficult than detecting its beginning. Tom constructs end with a closing brace, which is also the end-of-block indicator in our supported host languages. Detecting the end of a Tom construct may lead to what we call the island-island ambiguity, which happens between islands with different lengths. In this ambiguity, some closing braces have been wrongly interpreted as water. An example of a Tom program which contains this ambiguity type is given in Listing 12.

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

Listing 12: A simple Tom program which contains an Island-Island ambiguity

A GLL parser produces two different match constructs, and hence an ambiguity, from parsing the example of Listing 12. 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. We disambiguate this case by choosing the match construct which has well-balanced braces in its inner water fragments.

There are two ways to enforce the well-balancedness of braces. First, we can use a traversal function which counts the opening and closing braces inside water fragments of a Tom construct. This check is, however, very 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 Listing 3. For this, 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 a water elements. We chose the second option for solving the island-island ambiguity.

4.4 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. Hence, no ambiguity occurs at the beginning of a backquote term. However, a number of ambiguities might 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 Figure 5. For disambiguation, we prefer a `Variable` over `BackQuoteWater` by the rewrite rule of Listing 13.

```
{ PackedNode(x@["BackQuoteWater"]), PackedNode(y@["Variable"]) } -> { y }
```

Listing 13: Disambiguation pattern for `BackQuoteWater` and `Variable`

Another example of ambiguities in the backquote term can be observed in `‘x*`. This example can be recognised as a backquote term containing `VariableStar` or a backquote term containing `‘x` after which a water element (asterisk character) follows. The disambiguation in this case depends on the typing information of the host language. For example, the backquote term `‘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 directly, without any whitespace, comes after a variable, the recognition of `VariableStar` should be preferred. The SPPF corresponding to the parsing of `‘x*` is shown in Figure 6.

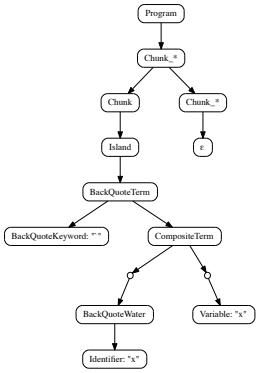

Figure 5: `BackQuoteWater/Variable`

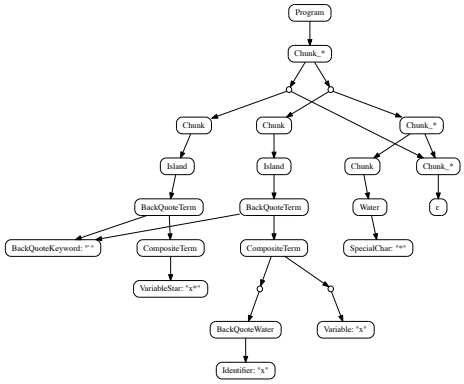

Figure 6: `Variable/VariableStar`

Note that in the SPPF of Figure 6 two ambiguities are present: one between the recognition of `x` as either `BackQuoteWater` and `Variable`, which can be resolved by the pattern of Listing 13. The new ambiguity caused by the presence of the asterisk is under `Chunk_*`. This ambiguity can be resolved by the rewrite rule presented in Listing 14.

```
{ x@PackedNode(["Chunk"("Island"("BackQuoteTerm")), "Chunk_*"(e)]),
  y@PackedNode(["Chunk"("Island"("BackQuoteTerm")),
                "Chunk_*"("Chunk"("Water"("*"), "Chunk_*"(e)))]),
  z* } -> { x, z* }
```

Listing 14: Disambiguation pattern for `Variable*` and `Variable`

An ambiguity similar to the one present in `‘x*` occurs in backquote terms which end in parentheses, see Listing 7. For example, in `‘x()` parentheses can be recognised as water and not as part of the backquote term. We use a rewrite 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. The rewrite rule for disambiguation of `‘x()` is exactly the same as the rule of Listing 14. The only difference is that `"Chunk"("Water"("*"))` should be replaced by `"Chunk"("Water"("(")` to detect the presence of an opening parenthesis after the backquote term.

5 Results

Using the island grammar and the disambiguation mechanisms presented in Section 3 and 4, respectively, we are able to parse most of the Tom programs from the tom-examples package. This package, which is shipped with a 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).

The parsed examples which have a size of about 10,000 characters can be parsed and disambiguated in less than one second, and the disambiguation time of an instance is always lower than its parsing time. Moreover, from what we have observed, the parsing time for Tom instances is linear.

Example file	#lines	#tokens	Parsing time	Disambiguation time	GLL 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

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

In order to evaluate the efficiency of our approach, we selected five representative examples: 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 that could be made as big as needed, by concatenating pieces of code.

Table 1 shows the statistics for parsing and disambiguating these examples using our GLL parser. We also compared our implementation with the current implementation of the Tom parser, which also uses an island-grammar approach, based on ANTLR parser generator. Currently, the ANTLR 3 based implementation is approximatively two times faster 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 future.

6 Related work

The name "island grammar" was coined in [19] and elaborated on in [11]. Moonen showed in [11] 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. The focus of [11] is more on reverse engineering and information extraction from source code rather than parsing embedded languages.

There has been a lot of work on embedding domain-specific syntax in general-purpose programming languages. For example, Bravenboer and Visser [4] 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 [7], 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 makes the disambiguation easier.

Developing a generic parsing method for island grammars benefits from generalised parsing, mainly because of the language composability and the support for full class of context-free grammars. ASF+SDF and the Meta-Environment have been used in a number of cases for developing island grammars. However, as shown in [13], the disambiguation mechanisms of SDF, *prefer* and *reject*, cannot deal with the ambiguities of island grammars in general. *Prefer* and *reject* identify which alternate of a nonterminal should be kept or removed respectively, by marking the nodes in the generated parse forest. However, the marked nodes are removed only if they are directly below an ambiguity node. This is not always the case, for example, when one of the nonterminals is nullable. Our motivation for using GLL and pattern matching was to not be affected by limitations of current toolsets, and have freedom in exploring different disambiguation techniques.

An example of using island grammars in parsing embedded languages which does not use a generalised parser is presented by Synytskyy et al. in [5,17]. 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[5], which is a hybrid of functional and rule-based programming styles. TXL uses top-down backtracking to recognise the input. In contrast to this work, we used a generalised parser and a set of term rewriting for disambiguation.

For island grammar-based parsing it is essential to deal with tokens which overlap or have different types. Aycock and Horspool in [8] 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 [20]. 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 about the type of a token based on the context in which the token's characters appear. As discussed in Section 2.4, our GLL implementation which uses a separate scanner does not require any specific treatment for tokens which overlap or have different types.

[4] Stratego home page. `http://strategoxt.org/`, Last visited: February 2012.
[5] TXL home page. `http://www.txl.ca/`, Last visited: February 2012.

7 Conclusions and Future Work

In this paper we presented how languages based on island grammars can be parsed using "off-the-shelf" 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 fragments. The primary challenge of island grammar-based parsing are the ambiguities resulting from the overlapping between host and embedded language tokens. We analysed the different ambiguity classes of Tom's island grammar and provided patterns for disambiguating them. Our approach can be applied to other island grammars as well.

Our main objective was to avoid modification of the parser in order to provide a generic solution for parsing embedded languages. Therefore, we proposed a method for disambiguating island grammars by adapting the grammar or post-processing the parse forest. In the case of Tom we used both: we adapted the grammar at one point, namely by introduced braces in the `water`. The other ambiguities where resolved via rewriting the resulting parse forest using the pattern matching mechanism of Tom.

We performed a number of experiments to validate our findings. In these experiments, we parsed a collection of more than 400 Tom files with different characteristics such as different sizes and number of islands. Our investigations showed that both parsing and disambiguation have a linear behaviour. Moreover, we compared our GLL implementation with the 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. Furthermore, our island based grammar is fully written in EBNF and supports modularity.

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 which traversal mechanisms can be applied to increase the efficiency of the disambiguation process. Finally, we plan to apply our approach to other language compositions, *e.g.*, Java and XML or Java and SQL, to determine how generic our method is.

Acknowledgement 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. E. Balland, P. Brauner, R. Kopetz, P.E. Moreau, and A. Reilles. Tom: piggybacking rewriting on java. In *Proc. 18th international conference on Term rewriting and applications*, RTA'07, pages 36–47, Berlin, Heidelberg, 2007. Springer-Verlag.
2. E. Balland, C. Kirchner, and P.E. Moreau. Formal Islands. In Michael Johnson and Varmo Vene, editors, *11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, pages 51–65, Kuressaare, Estonia, July 2006. Springer-Verlag.
3. M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. In *GPCE*, pages 3–12, 2007.
4. M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA*, pages 365–383, 2004.
5. J.R. Cordy. TXL - A Language for Programming Language Tools and Applications. *Electron. Notes Theor. Comput. Sci.*, 110:3–31, December 2004.
6. B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.
7. J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
8. j. Aycock and R.N. Horspool. Practical earley parsing. *Comput. J.*, 45(6):620–630, 2002.
9. M.E. Lesk and E. Schmidt. Lex - A Lexical Analyzer Generator. http://dinosaur.compilertools.net/lex/index.html, Last visited: June 2012.
10. M. Manders. mlBNF - a syntax formalism for domain specific languages, April 2011. MSc Thesis, Eindhoven University of Technology. `http://alexandria.tue.nl/extra1/afstversl/wsk-i/manders2011.pdf`.
11. L. Moonen. Generating robust parsers using island grammars. In *WCRE*, page 13, 2001.
12. P.E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76. Springer Berlin / Heidelberg, 2003.
13. E. Post. Island grammars in ASF+SDF, Summer 2007. MSc Thesis, University of Amsterdam. `http://homepages.cwi.nl/~paulk/theses/ErikPost.pdf`.
14. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, January 1992.
15. E. Scott and A. Johnstone. Modelling gll parser implementations. In *Proc. of the Third international conference on Software language engineering*, SLE'10, pages 42–61, Berlin, Heidelberg, 2011. Springer-Verlag.
16. E. Scott and A. Johnstone. GLL parse-tree generation. *Science of Computer Programming*, 2012. To appear.
17. N. Synytskyy, J.R. Cordy, and T.R. Dean. Robust multilingual parsing using island grammars. In *CASCON*, pages 266–278, 2003.
18. M. Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, 1986.
19. A. van Deursen and T. Kuipers. Building documentation generators. In *ICSM*, pages 40–49, 1999.
20. E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.