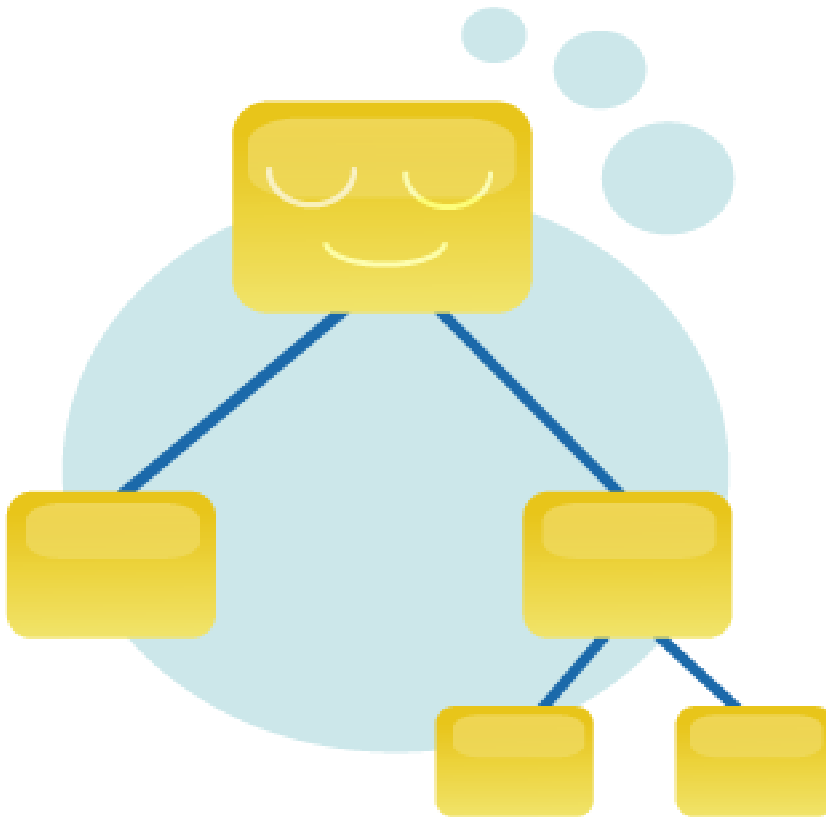
Island Grammars in ASF+SDF



Erik Post, 2007

.

Erik Post
epost@science.uva.nl
Summer 2007

"Don't let it end like this. Tell them I said something."

— last words of Pancho Villa (1877-1923)

Master's Thesis in Computer Science
University of Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Programming Research Group
Supervisor: Prof Dr P. Klint

.

This thesis is dedicated to the loving memory of my father.

Abstract

In this thesis we discuss the development, implementation and application of island grammars, implemented in the ASF+SDF Meta-Environments's Modular Syntax Definition Formalism (SDF). We provide an overview of related work on the subject, and attempt to answer the following questions, namely: how suitable are they, how may we develop them and what improvements might be made to ASF+SDF? In particular, we present some non-trivial example grammars, suggest new disambiguation constructs, and detail certain efficiency optimizations to the implementation of the SGLR parser.

Contents

Contents i

Preface iii
Foreword iv
Acknowledgements iv
Document structure v

1 Introduction 2
1.1 Preliminaries 2
1.2 Why generic language technology? 4
1.3 Why automated software analysis? 7
1.4 Why Island grammars? 10
1.5 Why SDF? 12

2 The ASF+SDF Meta-Environment 15

3 Related work 18

4 Research questions 20
4.1 Research Questions 20

5 Lexical analysis case studies 22
5.1 Introduction 22
5.2 Lexical analysis and regular expressions 22
5.3 Syntax recognition in GNU Emacs 23
5.4 Syntax recognition in KDE kate 24
5.5 Syntax recognition with Doxygen/Lex 26
5.6 Discussion 29

6 Simple C island grammar 32
6.1 Introduction 32
6.2 Requirements 32
6.3 Design 33
6.4 Results 35
6.5 Discussion 35

7 Island grammar for C function calls **36**
7.1 Introduction . . . 36
7.2 Requirements . . . 36
7.3 Implementation . . . 37
7.4 Results . . . 39
7.5 Discussion . . . 40

8 Fact extraction **42**
8.1 Introduction . . . 42
8.2 Method . . . 43
8.2.1 ASF extractor . . . 43
8.2.2 Java extractor . . . 45

9 Validation **48**
9.1 Introduction . . . 48
9.2 Method . . . 48
9.3 Results . . . 51
9.4 Discussion . . . 51

10 Implementation comparison **53**
10.1 Introduction . . . 53
10.1.1 Full Grammar . . . 53
10.1.2 Varieties of the function call island grammar . . . 53
10.2 Comparison . . . 55

11 Engineering advice **57**
11.1 Introduction . . . 57
11.2 Signal sequences and catchers . . . 57
11.3 A typical step in the design process . . . 58
11.4 Tolerance . . . 59
11.5 Correctness . . . 59
11.6 Compositionality . . . 60
11.7 Nested sorts . . . 60
11.8 Shortcuts . . . 61
11.9 Island/water ambiguities . . . 62
11.10Portability across languages . . . 63
11.11When to use island grammars . . . 65
11.12Miscellaneous points of advice . . . 66

12 Disambiguation **67**
12.1 Introduction . . . 67
12.2 Existing disambiguation mechanisms . . . 67
12.2.1 Prefer/avoid . . . 67
12.2.2 FilterPT . . . 69
12.2.3 Priorities . . . 69
12.3 Discussion . . . 70
12.3.1 A prefer-to filter . . . 71

13 Lexical inheritance analysis **72**
13.1 Introduction 72
13.2 Definitions 73
13.3 Example 74
13.4 Results 76
13.5 Discussion 77

14 SGLR performance improvements **79**
14.1 Introduction 79
14.2 Methods 80
14.3 Implementation 82
14.4 Results 84
14.5 Discussion 84

15 Proposed improvements to the Meta-Environment **88**
15.1 Information in MetaStudio 88
15.2 Suggested new features 90
15.3 Other 92

16 Conclusions **94**
16.1 Summary 94
16.2 Contributions 95
16.3 Future work 95

A Source Code **96**
A.1 Comment extractor source code 96
A.2 Function call extractor 102
A.3 Extractor using suckpt 108
A.4 SuckPT 110

B ASF+SDF Meta-Environment problems **114**
B.1 Meta-Environment/MetaStudio issues 114
B.2 Documentation issues 115

Bibliography **117**

Preface

Foreword

The occasion of finishing this thesis stirs in my breast the great desire to say many things. In this instance, however, my penchant towards being long-winded is inhibited my apparently even greater (and perhaps more detrimental) inclination for doing everything at the latest possible instance. Writing this foreword, I regret to note, is no exception, as the absolutely inert deadline for submission of this thesis is mere hours away. Nonetheless, one must endeavour to keep one's spirits up; Proverbs 17:28 tells us that "Even a fool, when he holdeth his peace, is counted wise: and he that shutteth his lips is esteemed a man of understanding." This provides at least some modest consolation (provided we overlook the more than hundred pages that follow). All in all, we can skip right to the acknowledgements without further ado. Enjoy reading!

— Erik Post, Wormer, 2007

Acknowledgements

I would like to thank the following people, listed, of course, in no particular order: My mom and dad, for going to great lengths to support me. (Some day I will pay back that laptop!) Wart van Zonneveld, for convincing me to do Computer Science instead of Dutch Literature. (I will have your head for that!) Bram Slinger, for all the great times, discussions, patience and music! Sanja Marusic, for always making me laugh, and for the chopsticks and Cat Stevens records and everything they represent. Anna den Enting for being her wonderful self, and for her love and support. Krista van Dale, for being a scholar and my guardian angel. Niels Lubbes, soon-to-be doctor of Computer Science, for having shared my predicament for a considerable time. Miss Bo van der Werff, for living in the same house with me for a couple of months that have come to mean very much to me (and for apparently not wanting to smother me with a pillow on account of them). Marcel Toele for ringing me up and scaring the bejesus out of me with new thesis deadlines every so often. Inge Huisman, for teaching me about singing and life. Simone de Vries for being the most loyal of friends one could wish for, and telling me to get to work. Paul Klint, for his sharp insights and the enthusiastic approach with which he manages to encourage people such as myself to (actually) get things done; and for his remarkable trait of managing to fit all of this and more into his tiny

handheld digital agenda. Jurgen Vinju, for helping and inspiring me practically every time our paths have crossed over the past 10 years. Brigitte at the Albert Heijn in Wormerveer, for her almost otherworldly radiant smile on the many days that she was (unknowingly) the only biped I talked to inbetween day-long development and (re)writing sessions. Theo and Maarten at Jottem and Stef at Business Event Studio. And of course Cat Stevens, Nick Drake, Deep Purple, Bach, Chopin and all the rest. To all of you, and to all the people who are not listed here and have put up with me locking myself inside my grotto for months (though some will have rather enjoyed the serenity of it all): thanks!!!

Document structure

The research to which this document pertains started out, after much deliberation, as an attempt to duplicate the functionality of Doxygen, a system for automatic generation of documentation from source code, in ASF+SDF. Paul Klint left it up to me to decide whether I would like to use full grammars or island grammars in doing so. I thought at the time that it might be a good idea to start out using island grammars to see what possibly interesting issues I would stumble upon, and to use full grammars to the extent that they were available. It soon turned out that the 'issues' concerning island grammars would consume most of my time. It was then decided to change the focus of the research to island grammars, and leave the Documentation issue as an aside.

Klint c.s. introduced the phrase 'grammarware engineering' as an emerging discipline within software engineering warranting attention for all kinds of reasons stipulated in [KLV05]. With this in mind, one of the aims of this research project is to familiarize the reader with the field of island grammar engineering, using SDF as a starting point. It does so by pointing to relevant research and tools, highlighting features and problems, and providing engineering guidelines for island grammar development.

In addition, common issues arising in island grammar development are discussed by means of examples, taken mainly from the development of an island grammar with the purpose of identifying function calls in C (see chapter 7).

Furthermore, we provide an in-depth discussion of improvements that might be made to the ASF+SDF Meta-Environment to facilitate the development and applicatin of island grammars. The chapters are structured as follows:

Chapter 1 Introduces various concepts and technologies and makes the case for using island grammars and SDF.

Chapter 2 Looks at a number of relevant components that make up the ASF+SDF Meta-Environment, especially from a command line perspective.

Chapter 3 Presents a survey of existing literature and technology relevant to the use and development of island grammars in SDF.

Chapter 4 Poses the research questions to be answered.

Chapter 5 Compares a number of existing lexical analyzers that are in widespread use, focusing on their metasyntax and its effects on such properties as

maintainability. These approaches are then compared to the island grammar approach.

Chapter 6 Introduces a preliminary island grammar used to extract comments, declarations and simple function calls from C code.

Chapter 7 Documents the development of a more complicated island grammar aimed at identifying more complex function calls from C.

Chapter 8 Details the fact extraction performed using the island grammar from the preceding chapter.

Chapter 9 Describes the validation of the facts extracted in the previous chapter.

Chapter 10 Compares a number of varieties of the function call island grammar developed in chapter 7.

Chapter 11 Discusses a number of observations and design issues that follow from them in a general way, and presents development advice on these issues where possible.

Chapter 12 Discusses island grammar-specific disambiguation issues and introduces a new type of construct to facilitate this.

Chapter 13 Introduces a method for analyzing some of the lexical properties of a language as they pertain to island grammar development.

Chapter 14 Proposes efficiency improvements to the Meta-Environment's parser (SGLR), which are specific to island grammars. The main idea is to compress irrelevant nodes in the syntax tree at parse-time to save space and improve parser performance.

Chapter 15 Suggests a number of general and island grammar-specific improvements to the Meta-Environment that might prove useful to grammar development in general and to that of island grammars in particular.

Chapter 16 Presents the conclusions of this research.

Appendix A Contains listings of source code to the programs and specifications developd as part of this research project.

Appendix B Discusses a number of problems encountered while using the Meta-Environment.

Chapter 1

Introduction

1.1 Preliminaries

Context-free grammar can be defined as follows [Sud97]:

Definition A **context-free grammar** is a quadruple $(V, \Sigma, P, \widehat{S})$ where V is a finite set of variables[1] or **sorts**, Σ (the alphabet) is a finite set of terminal symbols, P (the productions) is a finite set of rules such that $P \in V \times (V \cup \Sigma)^*$, and $\widehat{S}$ is a distinguished element of V called the start symbol. The sets V and Σ are assumed to be disjoint.

SDF, short for Syntax Definition Formalism, is a specification language for the definition of context-free grammars. [Vis97b] In other words, SDF is a syntax for defining syntaxes; this is often referred to as a **metasyntax**.

Productions are commonly written in the form $A \rightarrow w$, meaning that a variable A may be replaced by the **sentential form** $w \in V \times (V \cup \Sigma)^*$ in a derivation step. SDF uses a reversed notation: `w -> A` instead of $A \rightarrow w$ (note the different typefaces).

The set of context-free grammars is **closed under union**, also called **composition**. It is this important property that allows specifications written in SDF to be modular, meaning that an SDF specification may include another SDF specification.

Definition Given a context-free grammar G, the **language** of G, denoted $L(G)$, is the set $\{w \in \Sigma^* | \widehat{S} \stackrel{*}{\Rightarrow} w\}$. Here, the symbol $\stackrel{*}{\Rightarrow}$ denotes a finite number of applications of the production rules of G.

Definition A **construct of interest**, or COI for short, is a language construct we would like to recognize so we can perform analyses or transformations on it. We distinguish between constructs of **primary** interest, also called primary COI's, and **secondary** constructs of interest, aka secondary COI's. The former kind are the constucts we are interested in for their own sake. The latter kind covers those COI's that are not interesting per se, but because the recognition of primary COI's relies on them in some way. The constructs in which we are not interested are imaginatively called **uninteresting constructs**.

[1] Variables are also called *nonterminals*

Constructs of interest can also be classified as follows:

- A **valid surrounding construct** is a construct that may contain a certain other construct. For example, expressions can contain function calls. Therefore, expressions are valid surrounding constructs with respect to function calls.
- A **valid embedded construct** is a construct that may appear as part of another construct. Example: expressions may occur as the arguments of a function call. Therefore, expressions are valid embedded constructs of function calls.
- An **invalid surrounding construct** can *not* contain a given construct. Example: comments and strings cannot contain function calls, and are therefore invalid surrounding constructs for function calls.
- An **invalid embedded construct**, analogously, can *not* be contained within a given other construct.

Island grammars are *special purpose grammars*, that is: they are specified with a given task in mind, in the light of which certain constructs are constructs of interest and others are not. An island grammar will typically define primary COI's in terms of detailed productions called **island** productions or, simply, islands. The remaining constructs, including secondary COI's, are specified in as little detail as possible; preferably as **water**, whose syntax in most cases described lexically. That being said, secondary COI's may in some cases also have to be specified in such detail that they should be considered islands. a More formal definition of an island grammar is the following:

Definition An **island grammar** is a (possibly context-free) grammar $G = (V, \Sigma, P, \widehat{S}, W, I)$. Here, $W \subseteq V$ is the set of water sorts and $I \subseteq V$ is the set of island sorts. Membership of the sets W and I is given by the indicator functions $\chi_W : (V \cup \Sigma) \rightarrow \{1, 0\}$ and $\chi_I \rightarrow \{1, 0\}$

Specification of what the islands are and what the water is, i.e. defining $\chi_W(v)$ and $\chi_I(v)$, in an SDF grammar, may be done by:

1. Naming sorts accordingly
2. Tagging productions or sorts with island/water attributes
3. Injecting sorts into other sorts which are marked as islands/water

Note that the indicator functions $\chi_W(v)$ and $\chi_I(v)$ formalizing these notions do not rule out the possibility that a production or a sort is simultaneously water and island. This is useful for example if we want to ignore (i.e. treat as water) a given sort in some context, but not in others.

In contrast to island grammars, we also have **full grammars**, in which all constructs are specified in detail. Examples of this are the PICO and C grammars, which are distributed with the ASF+SDF Meta Environment.[2]

[2]Most island grammars mix a lexical and a syntactical analysis approach, with a preference for the former. Water is generally specified as lexical syntax and has, as such, no hierarchic structure. Islands are generally structured more like constructs in full grammars.

Suppose we had a language L, and we set out to create an island grammar G_i that accepts L. Due to the liberal way in which water productions are usually specified, $L(G_i)$ will generally be a strict superset of L. This property associated with island grammars is referred to as **tolerance**.

A string is said to be (syntactically) **correct** with respect to a grammar G or the language $L(G)$ if it can be derived from the start symbol of G in a finite sequence of applications of the productions in G. A parser for a context-free language L is said to **accept** a string s if $s \in L(G)$.

The notion of **correctness** with respect to an island grammar, in addition to its meaning as defined above, is defined as the degree to which it admits **false positives** and **false negatives** with respect to its purpose, i.e. the recognition of certain COI's. False positives are substrings that are unintentionally (thus, incorrectly, given the purpose of the grammar) identified as constructs of interest by the parser. On the other hand, **false negatives** are strings that are actually constructs of interest, but are not recognized as such (i.e. parsed as water) due to an error in the specification.

Island grammars can generally rely less on syntactic context than full grammars can for recognizing constructs. This is because the necessary detailed knowledge of a language's syntactic structure is simply not present in the island grammar. To compensate for this, we rely on certain lexical properties of our constructs of interest. **Signal sequences** are used for detecting constructs of interest based on characteristic sequences of characters they may contain. For example, a function call in C will contain parentheses, and a function definition in Python will contain the string 'function'. In order to prevent false negatives, we can exclude such sequences from water sorts, thus preventing the constructs containing them from being accidentally swallowed by overly tolerant water sorts. Excluding signal sequences from a water sort may initially cause parse errors if the parser encounters an occurrence of such a sequence in a term. To overcome this, we can specify **catchers**; productions intended for 'catching' signal sequences occuring outside of COI's. Examples of this can be found throughout this thesis, particularly in chapters 7 and **??**.

Island grammars are particularly suitable for **fact extraction**. Then, the island grammar is used to identify certain constructs of interest in a program text.

The constructs of interest can be *extracted* from the source code and, for example, stored in some representation relevant to the task at hand, thus obtaining extracted **software facts**.[3]

1.2 Why generic language technology?

Programmers find themselves faced with the role of translating real world knowledge and problems into representations that a computer can process. There used to be a time when the grass was green, and computer programmers earned their wages by plugging cables into switchboards in all sorts of bewildering patterns.

[3]An example of a relevant storage format would be to store function definitions and the calls occurring within them as tuples. Interpreted as an edge list, the vertices being the list of all functions defined and called in a program, induces a directed graph (the call graph). More on this in a subsequent section.

The subsequent development of punch cards — bits of cardboard that one punches holes into representing machine code instructions — was considered an important stride in usability. Nowadays, we have general-purpose programming languages such as C, Java, Python and Haskell, as well as domain specific languages (DSL's) which allow one to express a given problem to be solved in a form that is amenable to the specific *kind* of problem.[4] Table 1.1 lists some examples, most of them widely used.

problem domain	**DSL**
mathematics	Mathematica, MatLab, SciLab
chemical reactions	GasEl [BIK06]
music	abc
business processes	COBOL
formal grammars	SDF, GNU Bison, ANTLR
markup	HTML, LaTeX
graph drawing	dot, part of GraphViz

Table 1.1: Some examples of Domain Specific Languages

This diversity of languages is, compared to the punch card days, great for the end-user. However, while from his or her perspective we are moving ever further away from nasty low-level interactions of plugging cables around to represent the zeroes and ones of binary machine code instructions, at the end of the day we will still need to have instructions translated to a computer-processable format, if we want the computer to get anything done. Formal programming and specification languages are the lubricant in this human-computer mismatch.

Listing 1.1 Hello world in the C programming language

```
1  /*
2   * This program prints Hello World!
3   */
4
5  #include <stdio.h>
6
7  int main(int argc, char *argv[])
8  {
9    printf("Hello, world!\n");
10 }
```

The figures show the source code of a small program (listing 1.1) written by a human (i.e. myself) is translated to some intermediate formats (listings 1.2 and 1.3) by a compiler, before eventually ending up as a sequence of 'zeroes and ones'[5] that the computer can execute. Each intermediate representation format encodes at least the same information, possibly with some bits (such as the comments between /* and */) left out that may be uninteresting to the computer given its task of running the software.

So, in obtaining a machine-friendly representation of some problem from its

[4] It is a misconception that real hackers can sing baud straight into the mouthpiece.

[5] The figure corresponding to this representation is left to the reader's imagination.

human-friendly description, we encounter the task of language analysis, typically at many levels.

Formal languages allow us to express matters in intricate detail, which is required for many problems and, hence, their solutions. Consider a description such as: $(x+5)^3$. This is an example, and a very simple one, that immediately clarifies why it would be preferable to have a formal language, such as that of elementary algebra, so we can dispense with the very cumbersome and ambiguous: "Take x, then add 5 to it, then raise to the power of three". The analysis of natural language, written or spoken, involves automatic language analysis as well, which in many senses can be more complicated than their much more structured formal counterparts.

Listing 1.2 Hello world translated to Linux/x86 assembly language

```
        .file   "test.c"
        .section        .rodata
.LC0:
        .string "Hello, world!\n"
        .text
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
```

From the above, it follows that we need language analysis tools. Given the complexity of some of the problems in the domain of automated language analysis, we need strong theory, tools, paradigms, and generally anything we can get our hands on. There are many, *many* formal languages in day-to-day use at the moment. Some radically different from each other, while others are related dialects of a common language. Differences notwithstanding, the purposes and general structure of one language may be quite similar to that of others. This similarity can be exploited by reusing analysis technology for one language in analyzing others.

As an example, note that the assembly code in listing 1.2 is specifically generated for the Intel-designed x86 family of processors, which are to be found in most modern desktop and notebook computers. If we were to compile this program on a computer whose processor belonged to IBM's PowerPC or Sun's SPARC processor families, the generated code would look quite different (even though the structure would be similar). So, what goes into a language analysis

tool at the front end in one language (a program written in C, in our example), may come out at the back end in an different shape, depending on the purpose of the language analysis (here, compilation to a certain processor's machine code). Two separate subtasks emerge from this: language *analysis*, which deals with what a language looks like, and language *processing*, which in our example means translation to machine code.

Many existing language analysis tools[6] mix the description of what a language *looks* like (called *syntax*), with descriptions of what it means; what to do with it (called *semantics*). This makes a specification specific to both a given language *and* a task. Reading such specifications can be a bit like reading two books at the same time, and alternating between them every other sentence; hopelessly confusing. This makes it difficult to develop and maintain them, and to use them for other purposes than the one they were originally designed for.

Many tools, as if writing a language specification isn't challenging enough as it is, also lack features that enable reuse of specifications for languages that are similar.

The similarities noted before may be exploited by flexible, decoupled tools that separate such concerns as specification and processing, and provide features to reuse technology for similar cases. Technologies that do this are referred to as *generic language technology.*

Listing 1.3 Hello world in hexadecimal x86 machine code

```
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000
00000010: 0200 0300 0100 0000 9882 0408 3400 0000
00000020: 4807 0000 0000 0000 3400 2000 0700 2800
00000030: 1900 1800 0600 0000 3400 0000 3480 0408
00000040: 3480 0408 e000 0000 e000 0000 0500 0000
00000050: 0400 0000 0300 0000 1401 0000 1481 0408
```

1.3 Why automated software analysis?

> *"[The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."*
> — Edsger Dijkstra, *The Humble Programmer* (1972)

According to [Ben90], maintenance accounts for 70% of software budgets, software comprehension and the impact assessment of change requests being the two most expensive components [BA96]. How do we go about obtaining such understanding of software systems to be maintained? Let's try to answer this question based on the examples given before.

While the machine code of listing 1.3 is very useful in that it constitutes a working program that we can run on a computer, it is not the most suitable

[6] Lex, Flex, yacc, Bison, etc.

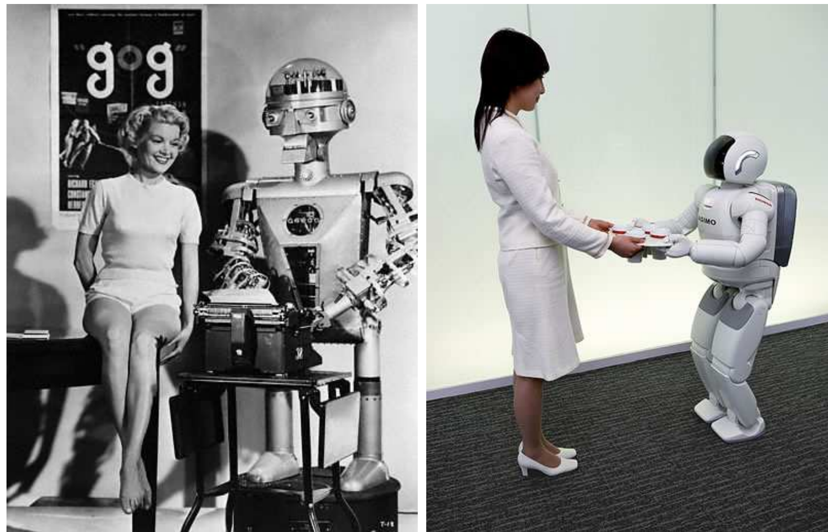


Figure 1.1: Robots in various stages of evolution and complexity. The suave one on the right can be assumed to be orders of magnitude more complex in terms of software, its source code running into hundreds of thousands or even millions of lines.

format for humans to comprehend. The C source code in listing 1.1 looks a lot better already, and some sense can be made of it even by non-programmers.

We can see line 9 saying `printf`, followed by the text we want to print. Such a thing as printf is called a *function* in the C programming language and in many other languages. Functions are bits of program code (subprograms) that carry out a task specified by the programmer. Whenever this task needs to be performed in the program, we invoke the function by making a *function call*, such as to `printf` on line 9.

Typical software systems, unlike this simple example, may consist of hundreds or even thousands of functions and function calls. Their interactions can be quite difficult to follow by just looking at the source code. When trying to understand such a complex system, a very common question is: what does a certain function, say `look_at(object)`, do exactly? Such questions are in the domain of *software comprehension*, also called program understanding. The question posed is really twofold: what sort of task does the function itself perform, and what is its place in relation to the other components in the system?

Such information can be expressed in *software models*. It can really help to look at visual representations of which function calls which other functions, in order get a feel for how a program is structured. Let's consider an imaginary robot brain as an example of a large and complex software system (see fig. 1.3). An example of a software model is a function call graph, which lists the functions in a system and tells us which (other) functions in the system they call (figure 1.2).

Software models are commonly produced by automatic analysis tools such as

function	**calls functions**
look_at(object)	turn_head()
	recognize(object)
talk()	think()
listen()	think()
recognize()	think()

Figure 1.2: Example software model: a function call table

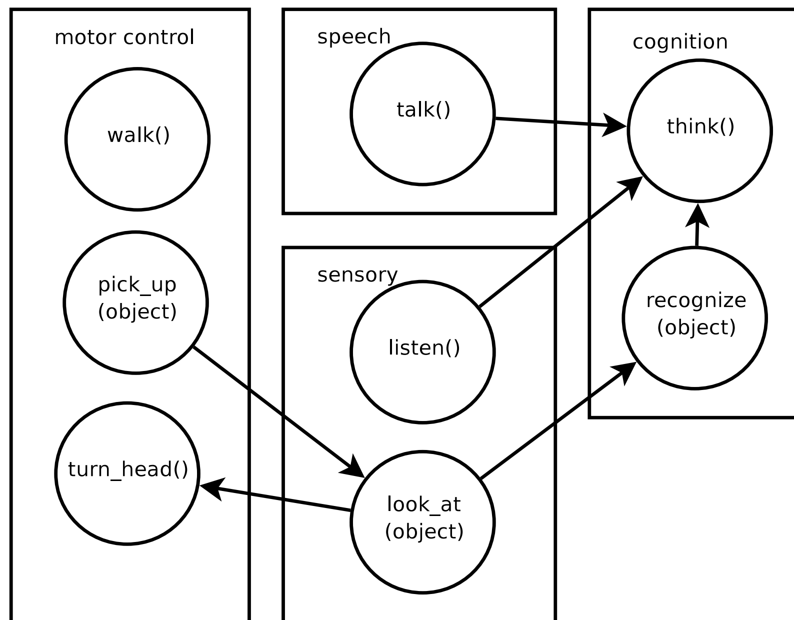


Figure 1.3: Component decomposition view of a robot brain

Doxygen[7] or Javadoc[8]. They analyze the source code and extract information (*software facts*) from it. They then present the extracted information in various forms; typically diagrams, tables and descriptions extracted from the code.

The extraction of facts from source code can be done in various ways. To make sense of it, the extractor needs built-in knowledge about what source code in the source language language looks like. This can be specified in the form of a *grammar*; a set of rules which describe exactly of what constructs a program in this language may consist. Such grammars are usually very lengthy and quite difficult to develop.

However, when we are looking only at a certain aspect of source code, such as the function call relationships it exhibits, there are many other details that we are *not* considering. Leaving out details about parts irrelevant to a given view of a system is called *abstracting away* of information. This is a central concept in software engineering, which allows humans to work on a complex systems consisting of many thousands of operations without going potty.

Since we are only interested in some parts of the source code, such as function

[7] `http://www.doxygen.org`
[8] `http://java.sun.com`

calls, when we are producing diagrams for it such as figure 1.3, analyzing the code requires only *partial* knowledge of the language that the code was written in. This is where *island grammars* come in.

1.4 Why Island grammars?

Island grammars, unlike full grammars, do not describe the *whole* language, with all of the constructs that may occur in it, in detail; they describe only a part of the language, namely those constructs that we would like to extract from the source code. In the robot brain example, this would amount to the (names of) the functions being defined and called. Therefore, it would make sense to define an island grammar that describes in (some) detail what function calls and definitions look like (these would be our our *islands*), so we can extract the names of function from them. We disregard the rest of the language as uninteresting lexical fluff made up of random characters that we do not care about (*water*).

Example island grammar Suppose we needed a grammar for the language spoken on a typical farm. An SDF specification of a full grammar describing this language could look something like this:

```
module FarmLifeFullGrammar

lexical syntax
  [b][a]+[h]                          -> SHEEPWORD
  [m][o]+                             -> COWWORD

context-free syntax
  (Sheepish | Cow | Pigese | Doggy)+  -> Conversation
  SHEEPWORD                           -> Sheepish
  COWWORD                             -> Cow
  "oink"                              -> Pigese
  "woof!"                             -> Doggy
```

A typical conversation in this language may look like this:

```
  oink oink
```

module	contains functions
motor_control	pick_up(object)
	walk()
	turn_head()
sensory	look_at(object)
	listen()
speech	talk()
cognition	think()
	recognize(object)

Figure 1.4: Software model showing which module contains which function

```
mooooooooooo
oink
baaah baaaaah
woof! woof!
mooo
```

Now suppose we were only interested in what sheep might say. We could define an island grammar to capture only their utterances:

```
module FarmLifeIslandGrammar

lexical syntax
  [b][a]+[h]                        -> SHEEPWORD
  ~[bah]+                           -> WATER

context-free syntax
  (Island | Water)+                 -> Conversation
  SHEEPWORD                         -> Sheepish
  Sheepish                          -> Island
  WATER                             -> Water
```

We can see that this grammar leaves out all the productions for the words we are not interested in; they are captured by the generic *Water* sort instead, which is defined as a series of any characters except those used in *Sheepish*. In a real-world example, this type of definition may save many productions and result in quite a concise and economical grammar. Moreover, the island grammar is *tolerant*: it allows a wider language than the one defined by the full grammar. As such, it accomodates for dialects, e.g. `muuuu` instead of `moooo` (Scottish cows) and even for constructs not found in the original language at all (e.g. `cluck`). Now let's take a look how the example conversation above would be parsed in both our full grammar, on the left, and our island grammar on the right:

```
<Pigese> <Pigese>                <WATER> <WATER>
<Cow>                            <WATER>
<Pigese>                         <WATER>
<Sheepish> <Sheepish>            <Sheepish> <Sheepish>
<Doggy> <Doggy>                  <WATER> <WATER>
<Cow>                            <WATER>
```

As we can see, the island grammar results in a much less explicit parse; it just identifies the constructs we need and 'ignores' the rest. Based on this minimal grammar, we could for example build a program that extracts what sheep say from a farm conversation. This principle also applies to programming languages.

Another Example Moving on to a perhaps more interesting example, we will take a look at some assembly language we might wish to capture grammatically. See listing 1.4 for an example. Line 9 contains the instruction `JSR`, which is short for 'jump [to] subroutine'. The instruction `JSR CHROUT` tells the computer to jump to the subroutine that starts at the memory address defined by the

sybol `CHROUT`. This is what a function call, referred to as ‘procedure call’ in assembly languages, looks like in Commodore 64 assembly. We can define an island grammar that identifies all `JSR` instructions and comments; see listing 1.5. This grammar is quite small, and leaves out all of the information that is redundant for our purpose, i.e. the recognition of procedure calls.

Listing 1.4 Commodore 64 assembly (NMOS/CMOS 6502 processor family)

```
1   ; Print Hello, World!
2
3            .word $c000          ; file header
4   *=$c000                       ; start address
5   MAIN:    LDX #$00
6   Loop:    LDA hello,x
7            CMP #$00
8            BEQ Out
9            JSR CHROUT           ; procedure call
10           INX
11           JMP Loop
12  Out:     RTS                  ; return from subroutine
13
14  hello:   .asc "Hello, World!"
15           .byt 13,0
```

Example applications We finish by listing a number of example applications of island grammars:

- Extraction of specific constructs of interest such as symbol references. An example of this is discussed in [Moo02].
- Extraction of code sections from mixed-language code such as HTML/JSP [SCD03] pages or COBOL/CICS programs [SSV02].
- Deriving call graphs, described in this thesis from chapter 7 onwards.
- Extracting class hierarchies.
- Import tree recovery from build systems such as make, the GNU autotools toolchain, Apache ant, cmake, etc.
- Simple pattern matching tasks.

1.5 Why SDF?

Performance SDF uses a scannerless GLR parser, based on an algorithm proposed by Tomita [Tom85]. Several experiments comparing LL(n), Early and GLR parsing have shown that ‘GLR is a far better choice for grammars with local ambiguities, such as island grammars’ [Lee05]. In chapter 14 we discuss this in more detail, and provide some performance improvements to the SGLR parser.

Application range SDF lends itself to the convenient specification of a wide variety of languages. It can accept the entire class of context-free languages, whereas many other parser generators are, by contrast, restricted to some subclass of context-free languages. Yacc for instance only accepts LALR(1) grammars.

Modularity As mentioned, SDF supports the entire class of context-free languages. Because this class is closed under composition, SDF specifications can be combined to form new (composite) specifications. SDF specifications can therefore be modularaized, and import other specifications, which greatly facilitates reuse and flexibility.

Separation of concerns An SDF grammar and the code that operates on the constructs recognized by this grammar are strictly separated. This results in a very clean, flexible and reusable grammar definition compared language definition formalisms such as Lex and Yacc (see also section 5.5).

As a result, we can more or less plug and play with modules such as Comments, Strings, etc. (An important question in the context of island grammars however, is to what degree we can make use of this modularity feature, since island grammars are not closed under composition; see the research questions in section 4.1.)

These advantages make the Meta-Environment particularly well suited for application in mixed-language environments: the combination of a single analysis backend with different language frontends (specified as SDF grammars) allows one to analyze or transform constructs of the same nature in source files across many languages.

Furthermore, the ASF+SDF Meta-Environment consists of a growing number of tools useful for reverse engineering, a notable current addition being RScript [Kli05], which allows us to query extracted software facts by means of a relational calculus.

Also, the Meta-Environment provides an IDE called MetaStudio, which makes experimentation and interactive development easier. The 2.x version is written in Java and offers syntax-directed editing, some nice tree visualization capabilities, and more.

These advantages do come at a cost however: parsing languages defined by SDF specifications using SGLR is inherently slower that the more 'simplistic' approaches, but often there is no real choice in the matter sinces the latter do not scale well to large, complex or heterogeneous problems, in which case having a somewhat slow solution may be a relatively small inconvenience.

Listing 1.5 Island grammar for procedure calls in NMOS/CMOS 6502 family of processors found in the Commodore 64, Atari and Apple II among others.

```
%% NMOS/CMOS 6502 assembly language

module IslandGrammarAssembler

imports Layout

exports
  sorts Program FunCall
  context-free start-symbols Program
hiddens
  sorts ProcCall IntCall JumpTarget
        P DIGITS HexInt WATER Water COMMENT
        ID LABEL Label

exports

lexical syntax
  [0-9]+                    → DIGITS
  ~[\ \t\n\;]+              → WATER
  [\;]~[\n]*                → COMMENT
  [A-Za-z][A-Za-z0-9]+      → ID
  [\ \t]*ID[\ \t]*[\n]      → LABEL


lexical restrictions
  DIGITS                    -/- [0-9]
  WATER                     -/- ~[\ \t\n\;]
  COMMENT                   -/- ~[\n]
  ID                        -/- [A-Za-z0-9]

context-free syntax
   Water                    → JumpTarget
  %% JSR: jump subroutine
  'JSR' caller:JumpTarget   → ProcCall {prefer}
  'JSR'                     → Water {reject}
  ProcCall                  → FunCall

  ID                        → Label

  WATER                     → Water

  LABEL                     → P {prefer}
  LABEL                     → Water {reject}

  Water                     → P {avoid}
  COMMENT                   → P
  FunCall                   → P {prefer}
  P+                        → Program {avoid}
```

Chapter 2

The ASF+SDF Meta-Environment

Introduction The ASF+SDF Meta-Environment is a platform for interactive program analysis and transformation. It features:

- A modular context-free syntax definition formalism called SDF
- A term-rewriting language called ASF (Algebraic Specification Formalism) which enables the analysis and transformations of parse trees and custom data structures
- An IDE called *MetaStudio* in which we can edit and test SDF and ASF specifications and terms
- A parser and a parser generator
- Many other tools

While most tools are accessible from the command line, the MetaStudio IDE hides all the command line nitty gritty behind a nice Java-based GUI, which couples the various components by means of the *ToolBus* interconnection architecture. The *ATerms* data format is used for efficient interchange of structured data. [BJKO00]. We concentrate on the command line view of the system here.

There may be several reasons for wanting to access their functionality through the command line. For instance, some of the tools listed here are not accessible through the IDE. Other features are buggy in the 2.0RC2 release when used via the IDE. Furthermore, the command line allows for batch processing and connection to external components.

Command line architecture Most tools will tell you their basic usage when invoked with the -h switch. They input from stdin and to stdout (and stderr) by default. The trees, which are represented as ATerms, can be displayed in textual format by using the -t switch with most tools, as applicable. For a summary of tool functionality, please consult the table of figure 2.2.

Some usage examples follow. Parsing the term "Donald Duck" over an SDF grammar *MyGrammar.sdf*:

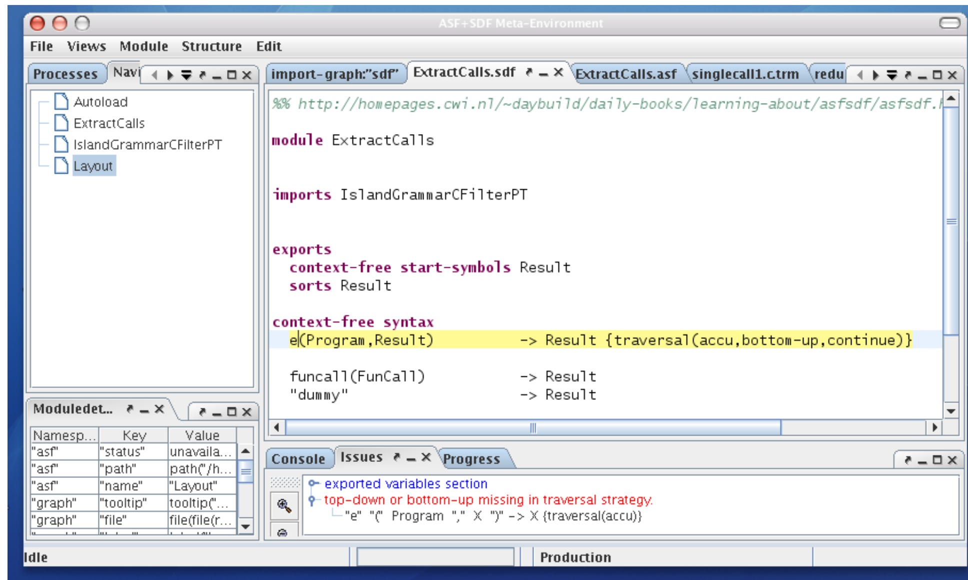


Figure 2.1: The MetaStudio IDE in the heat of the action

tool	**use**
pt-dump	Generates parse table from SDF
sglr	Parses input terms over a language, outputs a parse tree[1]
filterPT	Disambiguates by minimizing/maximizing occurrences of a given sort
addPosInfo	Adds position info (line, column) to all nodes in a parse tree
implodePT	Outputs trees in several levels of verbosity, e.g. ASTs
suckpt	Extracts subtrees from a parse tree[2]
asfe	Applies ASF equations to a parse tree

Figure 2.2: Some important tools in the Meta-Environment toolbox

```
pt-dump -m MyGrammar -o MyGrammar.tbl
cat "Donald Duck"| sglr  -p MyGrammar.tbl
```

Rendering a tree as a graph can be done as follows:

```
cat "Donald Duck" | sglr -p MyLanguage.tbl \
                  | tree2graph \
                  | graph2dot \
                  | dot -Tps > myPicture.ps
```

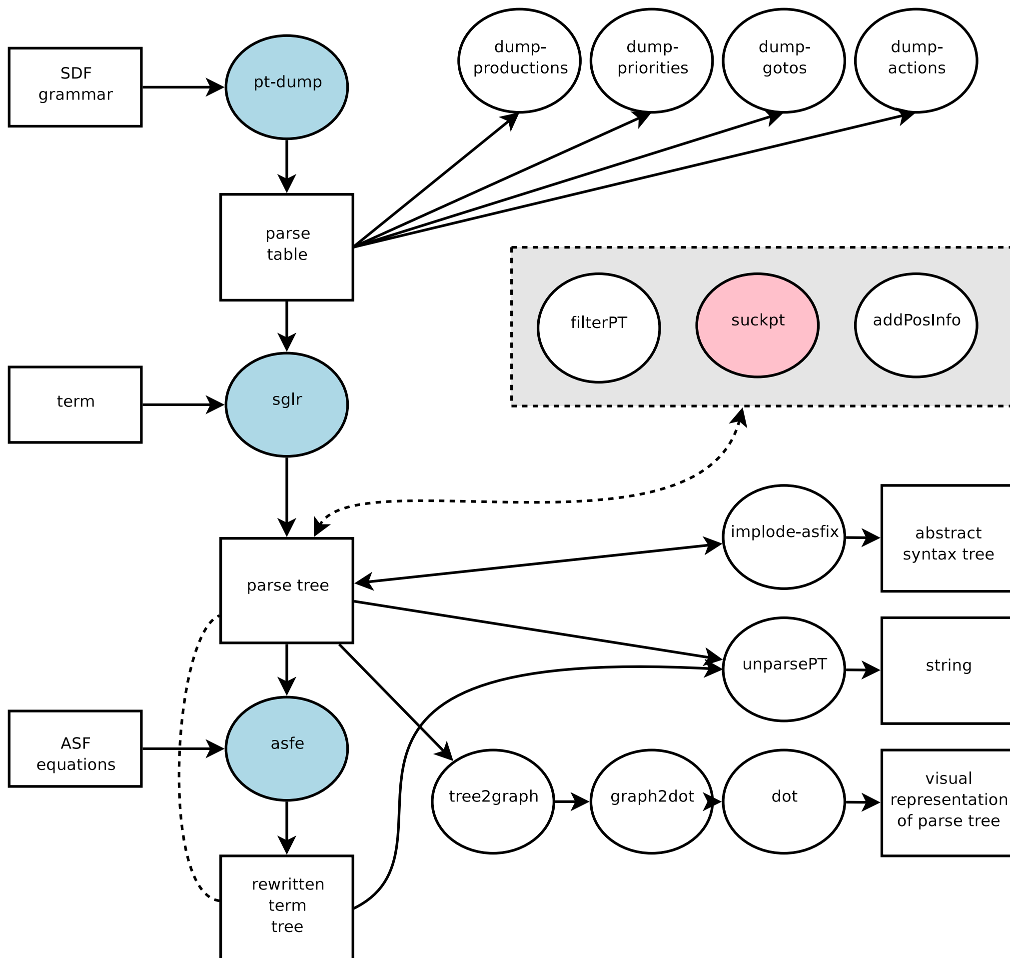


Figure 2.3: Architecture of Meta-Environment command line tools

Chapter 3

Related work

Much of the work on SDF island grammars focuses on COBOL. For example: [Moo02][KL03][?], etc. The Free University of Amsterdam (VU Amsterdam) has a research group that does a lot of COBOL research using ASF+SDF technology. The Software Improvement Group, a commercial spinoff from the CWI research institute co-founded by Paul Klint, is a name that also frequently occurs in relation to the topic [Ver00][Lee05].

In [Deu99] we can read about a 600 KLOC legacy COBOL system (*PensionFund*) that has been subjected to automatic documentation extraction. It is rather sparse in the example department, as it lists only a single 15-line SDF example. It mentions how island grammars facilitate parsing of vendor-specific language extensions.

Ernst-Jan Verhoeven has developed a COBOL island grammar in SDF [Ver00] that was meant to allow convenient source model extraction. It was intended as a possible replacement for the lexical analyzer used in the Software Improvement Group's SAT (Software Analysis Toolkit). This includes a tool called DocGen (short for Documentation Generator) which at that time used a lexical fact extractor written in PERL. Open questions: annotation with position information slows parsing down to a point beyond feasibility for real-time parses. How can this be remedied? He also points out some issues with ASF+SDF, and concludes that the COBOL island grammar that came out of his research still had too many problems to be used in the field.

Rob van der Leek has apparently picked up where Verhoeven left off. Among other things, he developed the tool `filterPT` as part of his MSc graduation research [Lee05]. I stumbled upon his thesis by accident via google, months after I had started developing an island grammar without his tool `filterPT` at my disposal. This tool is now part of the ASF+SDF Meta-Environment. In his thesis, he also discusses a hybrid lexical/syntactical approach to fact extraction called 'partial tokenization'. This is a multi-stage analysis. In the first pass, the source code is scanned lexically to locate possible constructs of interest, disregarding contextual information. In the second pass, the source code is again inspected using an island grammar to classify the context in which the suspected construct of interest appears, thereby eliminating false positives. Since the possible constructs of interest are tokenized in the first pass, the island grammar does not have to do this and will be less complex. He introduces ESDF, which amounts to SDF extended with a 'regular syntax' section, to facilitate

this partial tokenization.

He also discusses the use of Schroedinger tokens (which are not currently supported by SDF, which has a *scannerless* GLR parser). These are generated by a scanner encountering ambiguous tokens. These tokens lift some of the burden of lexical analysis from the parser. He concludes that the performance increase is a modest (constant) 10% and does not lead to a significant reduction in parsing complexity.

Fuzzy parsing is mentioned in [Kop97], which distinguishes two types. The first is a probabilistic technique from computational linguistics. The second is a method of parsing in which a parser gets its input from a scanner. The parser remains idle upon encountering, say, a water token. The token scanned, then, would not be included in the parse tree. An application of this, which I refer to as *partial parsing*, is discussed in chapter **??**.

The development and fundamentals of SDF, SDF2 and SGLR parsing are discussed in [Rek92] and [Vis97]. ASF, in conjunction with SDF, offers term-rewriting based analyses and transformation of source code. Generic type-safe traversal functions, a relatively recent extension to ASF, is discussed in [BKV02] and [Vis02]. In the paper [BKMV03], an application of ASF is discussed in the semantic disambiguation of parse forests. This allows us to rewrite ambiguity nodes in parse trees by means of the `amb()` constructor function.

It is difficult, if not impossible, to establish whether an island grammar performs exactly as intended. The behaviour we would like to see is for it to recognize all constructs of interest in the source code (no false negatives), and to identify no other strings as constructs of interest (no false positives). Klusener and Lämmerl [KL03] get around this problem by starting with a full baseline grammar for VS COBOL II, and using its productions for deriving island grammars (or tolerant grammars as they call it) from it as needed. In this way, they gain some of the benefits of using island grammars, such as tolerance and terseness, while being able to ensure their correct behaviour. (Of course, the absence of an available baseline grammar is often the problem that suggests using island grammars in the first place, but still their paper provides interesting insights into many relevant topics.)

Klint [Kli05] discusses a methodology and implementation for analyzing extracted software facts, called RScript. It offers a convenient abstraction of software facts in the form of so called Rstores, and allows one to process these using a relational calculus. The provided tooling is to become part of the ASF+SDF Meta Environment.

Another attractive set of tools for parse tree analysis and transformation is Stafunski [LV03]. From the paper's abstract: "Strafunski is a Haskell-centred software bundle for implementing language processing components - most notably program analyses and transformations. Typical application areas include program optimisation, refactoring, software metrics, software re- and reverse engineering. Strafunski started out as generic programming library complemented by generative tool support to address the concern of generic traversal over typed representations of parse trees in a scalable manner. Meanwhile, Strafunski also encompasses means of integrating external components such as parsers, pretty printers, and graph visualisation tools."

Stratego, which supports SDF through a toolset called XT, implements enables language analysis based on rewriting with programmable strategies. [BKVV06]

Chapter 4

Research questions

4.1 Research Questions

Island grammars, by any account, cannot be considered a celebrity subject software engineering. A Wikipedia search[1] on the topic returns a hit to the Ipswich Grammar School, but nothing particularly relevant to the search term. At the start of this research project, Paul Klint expressed his suspicion that their usefulness may be limited, especially with respect to their lack of closure under composition, but the subject was apparently still too scarcely researched to make such assessments with any degree of certainty.

Much of the prior research on island grammars and ASF+SDF, amongst which a number of MSc theses, has focused on the recognition of language constructs that are fairly easily defined, such as high-level constructs (e.g. function definitions in C) or constructs wit a flat syntactic structure (e.g. variable occurences). This thesis aims to investigate how suitable island grammars are for more involved tasks, such as the identification of nested or recursive constructs, often using C function calls as a representative example.

Also, some problems with ASF+SDF have been pointed out in the literature and by some of my own experiments, and I will attempt to present solutions to some of them.

1. How do SDF island grammars relate to existing research, methodologies and technologies.

2. How can we implement non-trivial construct recognition and fact extraction tasks using island grammars? To what extent are island grammars viable as a lightweight approach to grammar development?

3. What notations and techniques does ASF+SDF currently provide that may be of use in the specification of island grammars? What modifications or extensions would facilitate their development and application?

4. What can we do to improve the efficiency of parsing terms over an island grammar? What are possible optimizations? What language features may improve performance?

[1]Search was erformed on June 20, 2007.

5. Which factors contribute to the reusability of island grammars? Can we still write modular specifications, despite their lack of closure under composition? Do island grammars facilitate language-parametric fact extraction?

Chapter 5

Lexical analysis case studies

5.1 Introduction

In this chapter we shall take a look at some real-world examples of automated software analysis using approaches other than island grammars. This should help to answer the question whether there are any benefits to using island grammars, compared to lexical approaches. We will focus on their metasyntax, which is a key factor in the following areas:

- Correctness
- Robustness
- Maintainability
- Reusability

5.2 Lexical analysis and regular expressions

Context-free grammars allow us to specify recursion. This is needed to express sorts in terms of themselves, as is the case with arithmetic expressions for example:

```
context-free syntax
  Expr "+" Expr -> Expr
  Expr "-" Expr -> Expr
  Expr "*" Expr -> Expr
  Expr "/" Expr -> Expr
```

The specification of the sort on the right hand side includes itself in the left hand side. Lexical analysis is based on regular expressions, in which we cannot express recursion (or *nesting*) in such a way. To get around this, we would need to resort to code which explicitly maintains some state information. Lex allows us to include C code do such things. However, introducing general-purpose imperative programming language features into the equation, as opposed to declarative programming, reduces the maintainability and verifiability of a specification. The term ‘specification’ is perhaps also a little out

of place in this context, since such a specification is really more an imperative program with declarative elements.

5.3 Syntax recognition in GNU Emacs

Emacs is a text editor that has been around since the seventies. Its core functionality is written in C, mostly for efficiency reasons. The rest is written in a LISP variety called ELISP, short for Emacs Lisp. The name Emacs is an acronym of Editor MACroS, indicating the prominence of LISP ‘macros’ throughout the program.

We will look at how realtime syntax highlighting is achieved in ELISP. Emacs knows about a number of built-in syntax classes, such as:

syntax class	**denoted by**
whitespace characters	‘ ’
open parenthesis characters	‘(’
close parenthesis characters	‘)’
string quotes	‘"’
comment starters	‘<’
comment terminators	‘>’

We can use the following Elisp expression to mark “ ” as whitespace. (The question mark is used to indicate that the following character is the character to be put into the given class.)

```
;; Put the space character in class whitespace.
(modify-syntax-entry ?\  " ")
```

In addition to these syntax classes, there are a number of ‘flags’, which are really additional syntax classes:

syntax flag	**meaning**
1	1st character of a two-character comment start sequence
2	2nd character a two-character comment start sequence
3	1st character of a two-character comment end sequence
4	2nd character of a two-character comment end sequence

Most of these flags have been introduced to describe multi-character comment delimiters. An example of their use:

```
;; Mark ‘/’ as:
;; - a punctuation character,
;; - the first character of a start-comment sequence,
;; - and the second character of an end-comment sequence.
(modify-syntax-entry ?/ ".13")
```

Examples:

```
;; Put the space character in class whitespace.
(modify-syntax-entry ?\  " ")
     => nil
```

```
;; Make ‘$’ an open parenthesis character,
;;   with ‘^’ as its matching close.
(modify-syntax-entry ?$ "(^")
     => nil

;; Make ‘^’ a close parenthesis character,
;;   with ‘$’ as its matching open.
(modify-syntax-entry ?^ ")$")
     => nil

;; Make ‘/’ a punctuation character,
;;   the first character of a start-comment sequence,
;;   and the second character of an end-comment sequence.
;;   This is used in C mode.
(modify-syntax-entry ?/ ".13")
     => nil
```

An obvious problem with Emacs’ syntax recognition mechanism is that the notation is very complicated. This problem is exacerbated by the mechanism’s reliance on properties inherited from existing (standard) syntax descriptions. So in order to reliably define a new syntax recognition scheme in this way, we would first have to be able to understand (i.e. read) the existing ones.

In addition, the syntax recognition of several languages has problems with certain types of constructs, which is evident from incorrect highlighting. For example, the following PERL code applies a regular expression pattern match (the code `/["]/`) to a built-in variable.

```
if (/["]/) {
    ...
}
```

The double quote is apparently taken to be a string opener, since the code from the quote onwards is incorrectly coloured, which can remedied by putting additional double quotes in so that their number is even. This is typical of the kind of problems that regular expression matching has. Another example of the same nature is the use of (an uneven number of) dollar signs within a *verbatim* environment in Emacs’ Latex highlighting mode. The highlighter should ignore dollar signs appearing in the context of such the verbatim construct, but fails to do so because of its lacking awareness of syntactic structure.

5.4 Syntax recognition in KDE kate

Kate is a text editor that is part of KDE (the K Desktop Environment). This GPL-licensed project is, along with Gnome, one of the two main open source desktop environments in widespread use on Unix. With its 4.0 release planned for the autumn of 2007, which will run natively on Apple’s Mac OS X and Microsoft Windows, it is likely to become even more widespread.

KDE’s architecture is highly componentized, and the kate package provides text editing functionalities, amongst which syntax highlighting, to many other applications that make use of its library. KDevelop for example, a feature rich multi-language IDE for KDE, uses it as its default text editor component.

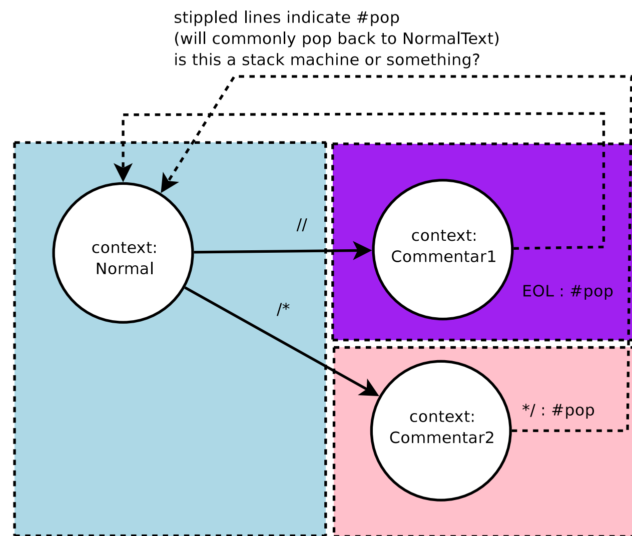


Figure 5.1: syntax highlighting in KDE's editor *kate*

Kate has separate syntax definition files for each language or language family. It uses these to highlight syntax and to enable code folding. Here, we will take a look at the file `cpp.xml`[1] displayed in part in listing 5.1.

Kate's C++ highlighter does not recognize and highlight function calls or expressions. Some excerpts of a Kate C++ syntax highlighting file follow. It has been edited for brevity and clarity in places.

First of all, we spot a variety of 'Contexts' with familiar names; Normal, Commentar1, Commentar2, Define. Let's take a look at the definition for the context *Normal* (lines 10-17). This is the default context; text is in the *Normal* context unless specified otherwise. It contains rules for deciding what to do when scanning certain characters. Line 11 says that if we scan a single '#'-character while in this context, we should change the current context to *Preprocessor*. A context change is achieved through the syntax `context="NameOfNewContext"` or `lineEndContext="NameOfNewContext"`.

Context can be nested; returning to the parent context is done by `context="#pop"`. Note the automaton-like behaviour. Also note that this notation allows for context-free behaviour, as contexts may be recursively nested.

We can more or less work out what's going on by studying the specification in listing 5.1 for a while, which is a big improvement over the Emacs case. It should be noted that syntax highlighting improves the readability of the above specification considerably. However, although the above type of specification may suffice for this particular type of task, being the syntax highlighting of C++ code, there are some inherent problems with its readability.

For example, state transistions, highlighting properties and language specification are intermixed. This lack of separation of concerns reduces readability. In addition, the specification is rather verbose. The specification for comments listed above, for instance, is only a part of what the entire specification con-

[1] Located in `/home/erik/.kde/share/apps/katepart/syntax/cpp` on my Kubuntu Linux system.

tains. If we look at a C/C++ block comment of the form `/* ... */`, called `Commentar2` in the specification above, we can see that the opening delimiter `/*` is specified in the context `Normal`, whereas the closing delimiter `*/` is specified in the context `Commentar2`. I have put them close together in the excerpt above, but they might be (and, in fact, are) quite far apart in the actual specification.

5.5 Syntax recognition with Doxygen/Lex

Doxygen is a program that can automatically generate documentation from source code. Documentation consists of hyperlinked descriptions of classes, methods, functions, etc, and of their interrelationships. This may be in the form of textual descriptions, call graphs, browsable module overviews, etc. Source code is annotated with special markup inside of comments. Such an annotated comment is bound to a construct immediately preceding or following it. In this example, the parameters of a C function declaration are annotated:

```
 /*!
  *  \param x base
  *  \param y exponent
  *  \return x^y
  */

int multiply(int x, int y);
```

Doxygen supports C, C++, Objective C, C#, Java, PHP and IDL natively, and some other languages through a mechanism of input filters. In order to parse languages with a non-C-like syntax, one has to resort to rewriters that preprocess the source code and transform it into a more C-like form. This can be quite complicated for languages with a radically different syntax (such as functional ones). This is one reason that Doxygen is not very useful for such languages.

Doxygen can export to e.g. Latex and HTML, as well as to the intermediate XML and perlmod formats. The XML format is intended as an intermediate representation from which all other types of output (HTML, Latex, DocBook, manpages, etc.) are to be generated, thus enabling the decoupling of the formatting of the documentation from its extraction. This is presently not the case, however. Generation of output formats is not done from XML, but is hardcoded directly into the output generator modules . The XML is available for developers to write custom tools around, but Doxygen itself does not make use of it (see figure 5.2).

Integration of these paths has been a stated development goal for some time, but is inhibited by several reasons; notably the lack of separation of concerns, which makes modularization hard. (For instance, part of the parsing happens in the documentation generators.)

The main part of Doxygen's language recognition and extraction, for *all* directly supported languages, is done in the monolithic 4997-line Lex specification `scanner.l`. It defines a scanner that analyzes the input files and produces an AST-like tree of entries containing properties of the constructs scanned.

The specifications for several languages are mixed into a single file, along with additional C++ code that allows recognition of more complicated con-

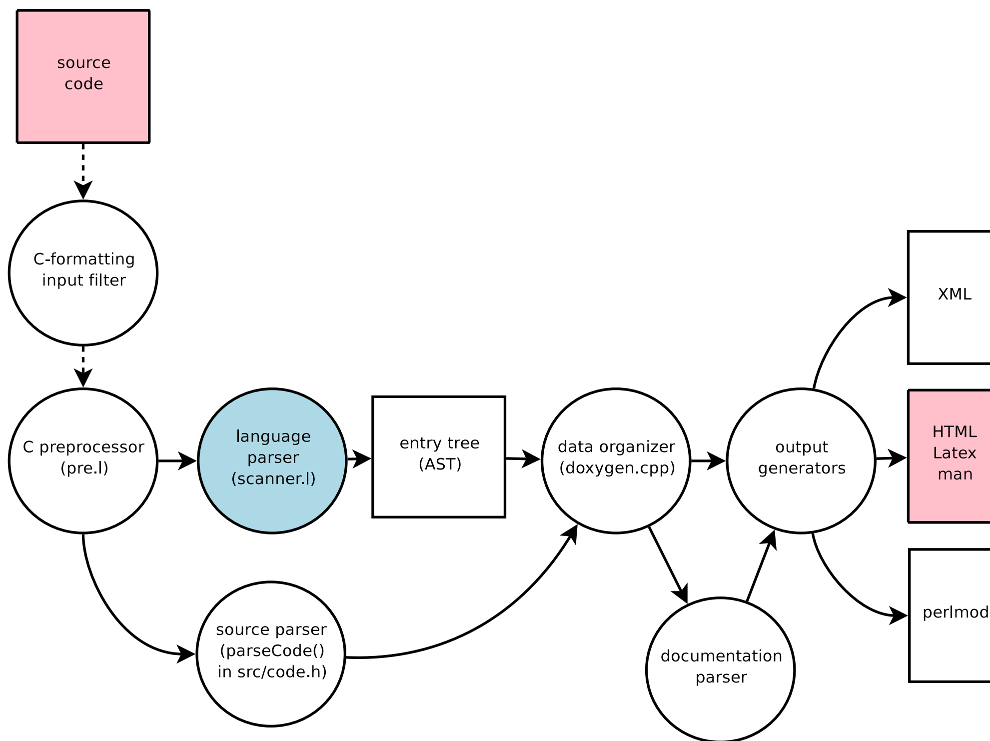


Figure 5.2: Simplified Doxygen architecture. The blue bubble is the monolithic parser. Input and output intended for the end-user is pink.

structs, such as K&R function declarations.[2] In addition, state information and updating of entries in the resulting syntax tree is also explicitly done in C++ code that is part the specification.

The following code excerpt from `scanner.l` shows how a fragment of the language definition (`<OldStyleArgs>[,;]`) is mixed with C++ code, which itself deals with multiple concerns at once. I have added indentation and comments for clarity:

```
<OldStyleArgs>[,;]
  {
    QCString oldStyleArgPtr;
    QCString oldStyleArgName;

    /* process K&R agruments */
    splitKnRArg(oldStyleArgPtr,oldStyleArgName);

    /* update AST information */
    QCString doc,brief;
    if (current→doc!=docBackup)
    {
```

[2]128 lines of C++ code in `scanner.l` are needed to deal with K&R style function definitions (the functions `checkForKnRstyleC()`, `splitKnRArg()` and `addKnRArgInfo()`).

```
    doc=current→doc.copy();
    current→doc=docBackup;
  }
  if (current→brief!=briefBackup)
  {
    brief=current→brief.copy();
    current→brief=briefBackup;
  }

  /* process K&R agruments */
  addKnRArgInfo(oldStyleArgType+oldStyleArgPtr,
                oldStyleArgName,brief,doc);
  current→args.resize(0);
  if (*yytext==';') oldStyleArgType.resize(0);
}
```

This is a comparatively simple excerpt from the Lex specification. Getting an idea of what the language to scan looks like by looking at this specification is rather a challenge. (The sparse use of comments does not alleviate this much.) Consequently, locating points where modification would have to be done, should the need arise, would be difficult; predicting what the effect of a given modification would be even more so.

Doxygen lists as possible improvements for future versions[3] [4]:

- Use one scanner/parser per language instead of one big scanner.
- Move the first pass parsing of documentation blocks to a separate module.
- Parse defines (these are currently gathered by the preprocessor, and ignored by the language parser).
- Add support to parse/document other languages. Suggested are: Perl, Ruby, Flex, Yacc, SQL, Visual Basic, Fortran, Matlab, Verilog, VHDL, Bash shell scripts.
- Add support for new output formats. Suggested are: XHTML, SGML, DocBook, Framemaker.

Since the language definition is hardcoded into the program in the form of a Lex specification, any additions or modifications to the recognized languages require at least a (re)compilation of the program. So, in order to extend Doxygen, one must be a developer with thorough knowledge of Lex and C++, and be familiar with the fairly obfuscated internals of `scanner.l` and the other Lex C++ files that constitute Doxygen; rather a lot to ask from would-be contributors, one might say.

The complexity and overlapping concerns explained above mean that Doxygen's language analysis functionality is difficult to maintain. It requires a lot of specific knowledge about the programming languages covered by it, at the very least.

If we compare this to an SDF specification, in which definitions for several distinct languages can be modularized, and the definition of the language can

[3] `http://www.stack.nl/~dimitri/doxygen/arch.html`
[4] `http://www.stack.nl/~dimitri/doxygen/todo.html`

be clarified by descriptive sort names, as opposed to obfuscated by interspersing it with Lex/C++ code, it becomes clear that SDF offers great advantages over Lex in this respect. Some documentation generators have already been produced using SDF, such as xDoc [Ver04], which uses SDF to generate language parsers, and Stratego/XT to perform further analysis.

5.6 Discussion

Lexical analysis is highly tolerant; whereas a parser needs to process the entire source text, lexical analysis will just happily detect the patterns it knows in the text without compaining about its well-formedness. This allows for features such as kate's almost instant syntax highlighting, even for large source files. In principle, we only have to highlight (up to) the visible sections of the code. Advantages notwithstanding, we have noted a number of drawbacks common to lexical approaches:

- The language specification's notation is often quite difficult to grasp (kate, Doxygen, Emacs)
- We must explicitly specify state transitions (kate, Doxygen)
- We must explicitly update additinal context-dependent variables such as insidePHP, insideJava, insideTryBlock, etc. (Doxygen)
- Hacks are required to deal with analysis of difficult constructs, such as K&R function declarations (Doxygen)
- Code that processes the tokens scanned is intermixed with the specification of the language itself (Doxygen).
- Updating of DFA state is intermixed with language specification (kate, Doxygen)

All in all, the developer is required to manually help craft the parse table when using these lexical tools, while SDF does this automatically, resulting in a much more readable (and therefore better maintainable) specification.

SDF is by definition about separation of concerns; it can describe the structure of a (formal) language, and nothing more. Processing its output (syntactically structured representations of source code) is left to other tools. Mappings from the parse tree or AST to software facts can be specified declaratively in separate components. In Doxygen, they are programmed imperatively by manipulating C++ data structures from inside the Lexer, such as Doxygen does.

The ability to connect powerful generic language analysis and transformation tools in a pipeline, such as SDF + ASF + RScript + some visualization component, provides us with the means for very powerful analysis or transformation across a wide range of languages. The increased proliferation of tools that support SDF as a language specification format, such as the Meta-Environment, Stratego/XT and Strafunski, etc, seems to spell a bright future for actual, working, widely available generic language technology tools. A tightly coupled monolithic and non-interactive system like Doxygen is bound to lose out against these, eventually. However, SDF and ASF have a number of issues, elaborated on in

chapter B.1, which should probably be solved before they can be expected to be generally taken up in the world outside of university-trained specialists. (There is a slight irony perhaps in the fact that, while Doxygen has had modularization of their parser component on their todo-list for quite some time, creators of such toolsets as the Meta-Environment and Strafunski rely on a mix of separate documentation tools (Doxygen, Javadoc and Haddock) themselves for generation of their API documentation.)

Listing 5.1 Source code excerpt of kate's C++ syntax highlighter specification in file cpp.xml

```
1  <list name="types">
2    <item> auto </item>
3    <item> bool </item>
4    <item> char </item>
5    <item> double </item>
6    ...
7  </item>
8
9  <contexts>
10   <context attribute="Normal Text" lineEndContext="#stay" ↙
         name="Normal">
11     <DetectChar attribute="Preprocessor" context="↙
           Preprocessor" char="#" firstNonSpace="true" />
12     <Detect2Chars attribute="Comment" context="Commentar 1"↙
            char="/" char1="/"/>
13     <Detect2Chars attribute="Comment" context="Commentar 2"↙
            char="/" char1="*" beginRegion="Comment"/>
14     ...
15   </context>
16
17   <context attribute="Comment" lineEndContext="#pop" name="↙
         Commentar 1">
18     <DetectSpaces />
19     <IncludeRules context="##Alerts" />
20     <DetectIdentifier />
21   </context>
22
23   <context attribute="Comment" lineEndContext="#stay" name=↙
         "Commentar 2">
24     <DetectSpaces />
25     <Detect2Chars attribute="Comment" context="#pop" char="↙
           *" char1="/" endRegion="Comment"/>
26     <IncludeRules context="##Alerts" />
27     <DetectIdentifier />
28   </context>
29
30   <context attribute="Preprocessor" lineEndContext="#pop" ↙
         name="Define">
31     <LineContinue attribute="Preprocessor" context="#stay"/↙
           >
32   </context>
33 </contexts>
```

Chapter 6

Simple C island grammar

6.1 Introduction

In this chapter, the development of a relatively simple island grammar for C is documented. Firstly, this serves as an introduction to the implementation of SDF island grammars, pointing out some common issues. Secondly, it provides a basis for the development of a more complex island grammar in the following chapter. As an example, we will take the following C source code:

```
/**
 * doxy comments
 */

int b;

int f()
{
    g();
    bla; /* this comment is ignored */
    h();
}
```

The idea is to extract the comments, declarations and definitions, which are prerequisites for automatically generating documentation from the source code.

6.2 Requirements

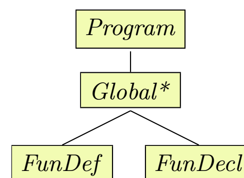
- Parse ANSI C
- Identify <u>comments</u> as constructs of interest
- Recognize function declarations and definitions (ANSI C style, no K&R)
- Recognize stand-alone, unnested function calls
- Parse but ignore global variable and type declarations

- Allow unambiguous parse by SGLR, without the need for external post-parse filters

Preprocessor directives are ignored. Strictly speaking, they are not part of the C language itself; they are expanded by a macro preprocessor, and this expanded form of the source code (without preprocessor directives) is fed to the compiler.[1]

6.3 Design

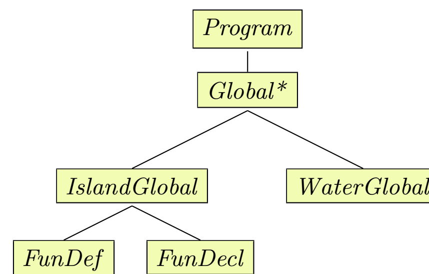
High level structure The top level syntax of a C program is the globals section. This section contains the function declarations and definitions we aim to recognize:



These constructs typically look something like this:

```
...    Id "(" ParamList ")" ";"             -> FunDecl
...    Id "(" ParamList ")" CompoundStat    -> FunDef
```

The sort *CompoundStat* represents the ‘compound statement’, also called ‘block statement’. Loosely speaking, it is a list of statements enclosed in curly braces “{” and “}”. We now define a water sort so that we may ignore whatever is in between function definitions and function declarations in the globals section. This gives rise to the following basic grammar structure:



Statement separation Consider again the basic form of a function definition given above. In order to parse a function definition, we must know where its opening and closing braces are. However, compound statements may also *contain* braces, for example as part of a string, array initializer, or nested compound statement. This in turn necessitates the definition of some of the internal

[1] Individual compilers may deviate from this scheme of dealing with preprocessor directives, as some directives may be used to pass arguments to the compiler.

structure of compound statements, which as we noted are basically made up of statements.

language construct	terminated by
statement	';'
compound statement	'}'
goto label	':'
`case`, `default` (part of `switch`)	':'
comment	'*/' or newline

Table 6.1: Statement terminators in C

Most statements in C are terminated by a semicolon ";". However, any of the constructs from table 6.1 may occur as part of a compound statement, and not all of them end on a semicolon. Therefore, we cannot simply specify a compound statement of code as being a list of statements separated by ";" (a semicolon). To help ensure the proper recognition of individual statements, we consider the statement separator ";" to be a **signal sequence**, a characteristic local syntactic feature of statement separation. In order to ensure that it is not accidently recognized as water, we explicitly exclude it from the definition of the sort *WaterStat*. Simple statements are covered by the production:

```
WATER* ";"                              -> WaterStat {prefer}
```

C has a number of ways in which a compound statement can occur: as autonomous statements, or as an embedded part of other statements, such as if-else blocks. We can circumvent the need to explicitly specify all such statement types (`if`, `for`, `while`, `switch`, `do ... while`, etc.) in detail by defining a generic statement:

```
WATER* CompoundStat                     -> WaterStat {avoid}
```

With this single generic production, an if statement will be parsed as follows:

The production also covers constructs more complicated constructs such as `do {...} while (...)`, and as such saves a lot of specific productions. Here is an example of how it breaks such statements down into separate *WaterStat*'s:

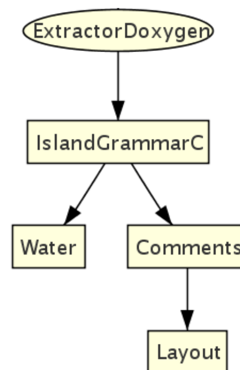
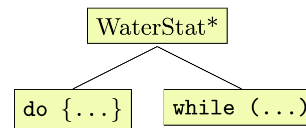


Figure 6.1: Import tree for the comment island grammar

```
do {
    ...
} while (...)
```



6.4 Results

The source code for the extractor is listed in section A.1. Application to the example C code given in the introduction section of this chapter results in the following output (which based on Doxygen's XML output format:

```
<comment>
  * doxy comments
</comment>
<memberdef kind="variable">
    <type>int </type>
    <name>b</name>
</memberdef>
<funbody>
    <funcall caller="g">g</funcall>
    <ignoredstatement />
    <funcall caller="h">h</funcall>
</funbody>
```

6.5 Discussion

We have seen that we can specify *separate* water sorts on different levels of the the grammar's syntactic hierarchy. This isolation reduces the risk of possible clashes, confining the effects of water sorts to their respective modules.

This increases the possibilities for compositionality of the grammar, a property discussed in more detail in chapter 11.

On the level of statements, we have seen how signal sequences (i.e. the semicolon) were used to help enforce the recognition of individual statements. This was done by excluding the signal sequence from the water sort *WaterStat*. This will cause the parser to generate a parse error upon encountering such a sequence, unless it occurs as part of a specified construct (i.e. island).

Chapter 7

Island grammar for C function calls

7.1 Introduction

This chapter deals with a more complex example of an island grammar, one that identifies function calls in C. As we argued in the introductory chapters, the function call extraction has many useful applications, their visual rendering as call graphs being just one. Also, in order to get at the function calls in a C program, we must descend several levels of nested constructs, which makes the problem interesting from a syntactic analysis perspective. The design and development of a function call grammar poses a number of questions fundamental to the topic of developing island grammars. To name a few: what constructs will we define in detail, and which ones do we leave as water? How can we make constructs of interest stand out from surrounding water?

7.2 Requirements

If we want to be able to determine not only which functions are called, but also which functions call them, we will have to define function definitions (*FunDef*) in such a way that we can tell from which function a given function call stems. Function definitions and declarations are also used to validate the extracted calls later on. Our requirements then are:

- Identify function calls
- Identify function declarations and function definitions
- Recognize ANSI C style function declarations and definitions (no K&R)
- Definition should be minimal in terms of the number of explicitly defined constructs and complexity and (less importantly) in terms of number of productions, number of sorts and lines of code.
- Preferably, allow unambiguous parsing, if possible without resorting to post-parse filters external to SGLR

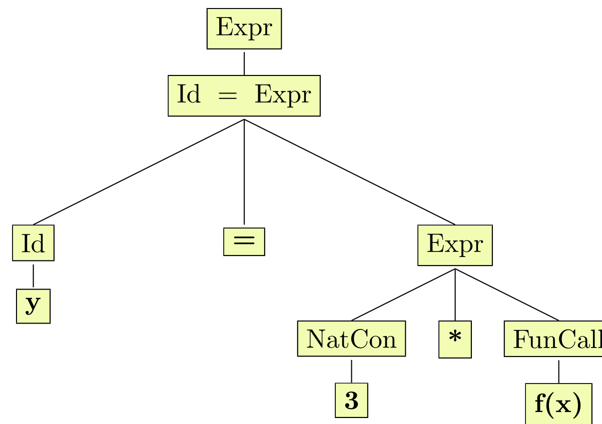


Figure 7.1: Parse tree of an expression containing a function call

7.3 Implementation

Function declarations, which are mapped to the sort *FunDecl*, and function definitions, mapped to *FunDef*, are both top-level constructs; they occur in derivations from the start symbol *Program* within a few derivation steps. Their specification is fairly straightforward, and can be found in the source code in section A.2.

When extracting function calls, we must ask ourselves: in which contexts can function calls (the sort *FunCall*) occur?

1. As a stand-alone statement (the sort *Stat*)

2. Inside (non-constant) expressions (the sort *Expr*)

Recognition of a *FunCall* as a stand-alone statement among other statements is not particularly difficult, given a correct definition of *FunCall*, which is also straightforward:

```
Id "(" {Expr ","}* ")"    -> FunCall
```

In fact, (1) is merely a special case of (2) since an expression by itself, and consequently a function call too, constitutes a valid statement. The second case is quite involved however.

Although we are not interested in expressions per se, they are important in the sense that they are a context in which function calls might appear. An example of this is the expression: `y = 3 * f(x)`. A typical parse tree for such an expression over a full grammar is depicted in fig. 7.1. Expressions themselves may occur in many contexts; the table in figure 7.1 lists some of them. There exist a number of possible strategies for recognizing function calls. In decreasing order of structural preciseness, we have:

1. Recognize function calls as part of *Expr*, where *Expr* may occur only in prescribed contexts, such as assignment statements.

2. Recognize function calls as part of *Expr*, where *Expr* islands may be randomly scattered through function bodies otherwise defined as water. In particular, statements are not recognized as separate entities.

3. Recognize (compound) statements as composed of either water or *Expr*, in which *FunCalls* may reside.

4. Recognize function calls anywhere, except in explicitly defined contexts in which they may *not* appear, such as comments and strings.

Stat	**contains nested Stat?**
if (***Expr***) *Stat*	yes
else if (***Expr***) *Stat*	yes
while (***Expr***) *Stat*	yes
Id = ***Expr***	no
Expr ? *Expr* : *Expr*	no
return ***Expr***	no

Table 7.1: Some contexts in which *Expr* may occur

Approach number (2) is risky; if *Expr*'s are not sufficiently well-defined, the parser may interpret something that is supposed to be an *Expr* as water instead. Suppose for the sake of argument that we had accidentally omitted the rule `Expr "?" Expr ":" Expr -> Expr`. Now, our parser would fail to recognize any subterm containing this operator as an expression. This implies that any *FunCalls* occurring in our unrecognized *Expr* will also not be properly recognized and, hence, will be missed by the extractor. So, approach (2) may result in false negatives.

Another argument for defining *Expr*'s explicitly: when recognizing function calls, we would like to know at what symbol the *FunCall* starts, but also where it ends. In C, they end on `)` (a closing parenthesis). However, arguments to a function may be complex expressions themselves, consisting of string literals, bracket expressions, typecasts and whatnot. In particular, they may contain commas and parentheses, which complicates the recognition of the *Expr*'s in our definition of *FunCall*, and therefore of *FunCall* itself:

```
myfun("(\(()(\")");    /* which ")" closes the FunCall? */
```

So *Expr*'s are valid *embedded* constructs with respect to *FunCall*'s, and in order to recognize them we must consider all these factors. Here are some examples of invalid surrounding constructs:

```
char *s = "myfun()";   /* string, not a FunCall  */
/* myfun */            /* comment, not a FunCall */
```

These constructs have been defined in the island grammar through productions of a chiefly lexical nature. Our island grammar relies on string literals to disqualify certain substrings as possible *FunCall*'s, and to swallow parentheses not belonging to a function call.

The implementation of approach (3) is discussed in section 10.1.2. Regarding (4), it can be said that specifying the locations of possible function calls by

defining where they can *not* occur runs somewhat counter to the customary ways of thinking about language design. Grammars of this type are sometimes called *lake grammars*, and even though the approach is interesting, it has been left unimplemented due to time constraints.

In order to be able to properly handle all of the issues mentioned in an intuitive manner that also provides some confidence in the correctness of the resulting grammar, I have chosen to start out by implementing approach (1). That means we will only recognize function calls as part of *Expr*'s, and *Expr*'s may only occur in prescribed valid contexts.

Signal sequences As in the previous chapter, we distinguish between several source code levels, each of them with their own water sort. On the top level of globals and function definitions, we have *WaterGlobal*, which excludes semicolons as signal sequences. At the function definition level, inside statements, our signal sequences are semicolons, parentheses and curly braces, which are excluded *WaterStat* as such.[1]

Restricting the effects of water sorts to their own modules increases the compositionality of the grammar. It allows us to reuse, say, the module *Stat*, without the risk of its water interfering with the rest of the grammar, provided it is defined sufficiently restrictively.

Moreover, by preventing anything containing these characters from being parsed as water, we guarantee that we won't miss any possible function declarations, definitions or calls. Thus, we favour false positives over false negatives; the former can always be eliminated using semantic analysis, whereas the latter amounts to irrecoverable information loss in the parsing stage. In doing so, we also favour parse errors over false negatives, or to put it another way, correctness over tolerance.

Function and method calls have similar syntax across many procedural, functional, and other language families (C, Java, PHP, BASIC, Pascal, Python, etc):

```
f(..., ..., ...)
```

That is, an identifier followed by an opening parenthesis, followed by a comma-separated argument list, followed by a closing parenthesis. However, there are also languages such as LISP, Haskell or Matlab, which do not require parentheses for function calls in all cases. So by disallowing parentheses and such, we also favour correctness over language portability.

7.4 Results

The source code is listed in section A.2. We have defined the following constructs:

A simplified sorts graph of this grammar can be found in figure 7.2. Some notes on this grammar:

[1]Excluding signal sequences from water sorts is especially useful during development, since it alerts us to possible false negative in the grammar. At the end of the day, we will in principle have explicity defined all constructs that may contain signal sequences in the form of islands or catchers. While this may still allow for ambiguities around constructs concerning these signal sequences, false negatives due to excessively tolerant water sorts are at least ruled out. See also chapter 11.

Islands	**Water**
function declarations	builtin types
function definitions	typedefs
function calls	enums
expressions	structs, unions
statements which might contain expressions*	
A handful of keywords, mainly for rejection purposes	
goto labels, case statements, etc.	

*) if, else, return, etc.

- The definition of the *FunCall* sort is reused to capture function attributes in *FunDef*, which look something like this:
  `extern void *sbrk (intptr_t __delta) __attribute__ ((__nothrow__))`
  This trick seems to work; very agile indeed.

- One of the two types of function pointer calls, which looks like this: `(*fptr) (x)`, is not recognized, but may be easily added because the syntax is very similar to that of ordinary function calls. The other notation has exactly the same syntax as a regular function call and is therefore recognized properly.

- Functions returning pointers are not recognized.

- The C-builtin `sizeof()` is recognized as a function.

- C++ and Doxygen type comments are currently broken: they cause parse errors, while the island grammar from the preceding chapter did not.

7.5 Discussion

In order to be able to extract function calls, we have opted to define expressions, strings, etc, to a fairly precise level. Water is used to capture such constructs as function attributes and global variable declarations. In particular, structs and unions are not explicitly defined, but we do have a basic definition for variable declarations, required because they may contain expressions. It would not be much work to extend this to cover all possible types of declarations in C. So, even though this should strictly speaking be considered water, it is *almost* a precise definition. The same applies to other water constructs used in our grammar. They would not be very difficult to factor out, and indeed we have come close to defining a full grammar for C.

The expression grammar is considerably large, and has been a focal point of our development and disambiguation efforts. Some shortcuts were taken in defining it. In particular, the priorities and associativities defined are partly random and nonsensical in an artihmetical sense, and are intended purely for disambiguation. Moreover, expressions are not type-aware in any way, meaning that we can add integers to strings, and so on. Also, the definition of certain constructs such as the array indexing operator are more liberal than in ANSI C.

It has to be emphasized that a lot of the 'shortcuts' used, especially in the expression grammar, caused parse errors and (unacceptable) ambiguities later

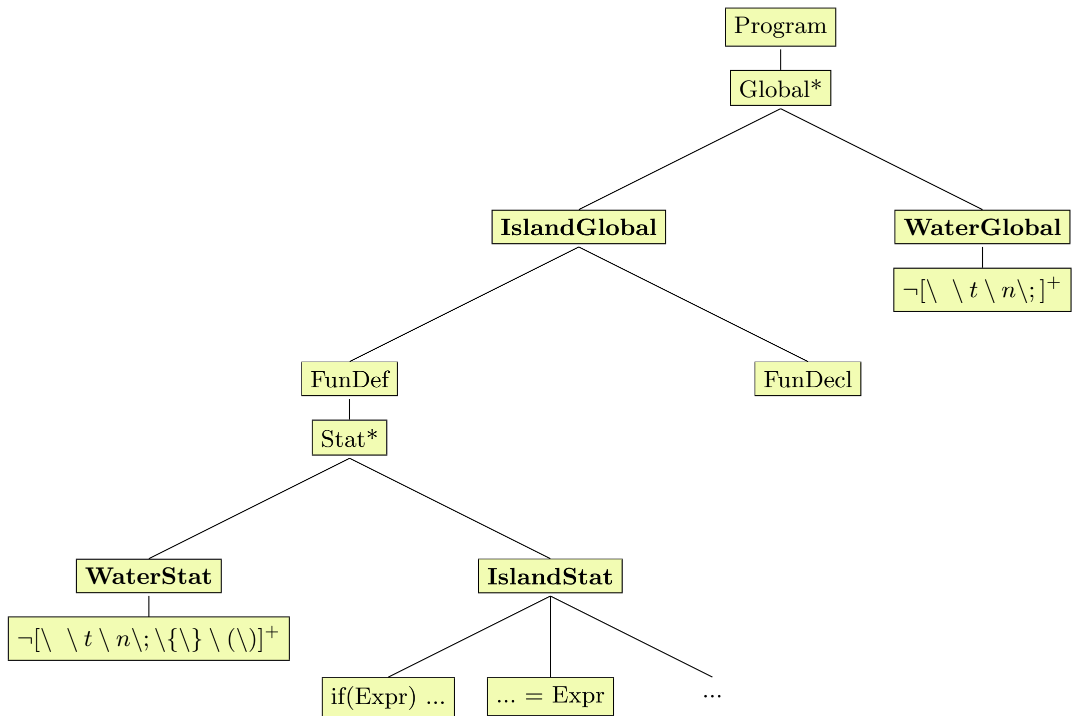


Figure 7.2: Simplified grammar tree for the function call island grammar, showing different water regions. *Expr* and *FunCall*'s inclusion is cyclic (not shown).

on that would have warranted being more precise from an early stage, since eliminating some of them turned out to be quite time consuming. This demonstrates some of the difficulty of predicting the benefits of a 'watery' approach to defining particular constructs.

Chapter 8

Fact extraction

8.1 Introduction

This chapter explains how the required software facts may be obtained from source code using the grammars developed in the previous chapters. It outlines a number of possible extraction toolchains using various technologies, and describes two of them in detail, namely one using ASF and another using a custom Java extractor called *suckpt*.

Marking parse tree nodes for extraction SDF provides a number of constructs that may be used to mark subtrees intended for extraction. Consider the following SDF production:

```
fname: Id "(" args: {Expr ","}* ")"  -> FunCall {cons("Call")}
```

Given this rule, we could extract subtrees from the parse tree or AST based upon sort name at the rhs (*FunCall*), the constructor function name (*Call*), or one of the labels (*fname* or *args*). The recent addition of labels to SDF allows us to refer to subsorts in a production by name, as opposed to, say, their order of occurrence within the production. All of this provides us with a number of convenient ways to perform language-independent extraction.

Subtree extraction There are several ways of extracting the relevant information from a parse tree in ATerm format:

- $\rightarrow$ parse tree (XML) $\stackrel{XPath}{\longrightarrow}$ calls (XML) $\stackrel{xsltproc}{\longrightarrow}$ HTML/DocBook/Latex/etc
- $\stackrel{ASF}{\longrightarrow}$ calls (ATerm) $\rightarrow$ calls (XML) $\stackrel{xsltproc}{\longrightarrow}$ HTML/DocBook/Latex/etc
- $\rightarrow$ AST $\stackrel{Stratego/XT}{\longrightarrow}$ calls
- $\stackrel{ApiGen}{\longrightarrow}$
- $\stackrel{suckPT}{\longrightarrow}$ calls
- $\stackrel{Sdf2Haskell}{\longrightarrow}$ AST (Haskell)$\longrightarrow$ calls

The XPath approach is used in [Lee05]. Stratego/XT [BKVV06] is used in xDoc [Ver04]. The *Sdf2Haskell* tool is part of the *Strafunski* toolset [LV03]. There also exists a Java visitor generator called JJForester, which can be used to specify traversals over parse trees in Java.

We will be using *suckpt*, a program which has been developed in the course of this research project and described in chapter 8.2.2. Using it, we can extract subtrees from a parse tree based on the sort at the right hand side of their top node.

8.2 Method

8.2.1 ASF extractor

Parse tree traversal by explicit term rewriting We can extract software facts by parsing a source code text, traversing the resulting parse tree, and extracting the desired information from the relevant nodes. We may also take contextual information into account involving other nodes which are in some way related.

ASF allows one to specify type-preserving transformations on parse trees. We can do so by writing explicit equations for each type of node we would like to traverse. For example, to traverse the parse tree of figure **??** in order to reach a function call, we must must first traverse nodes of type *Program*, *FunDef*, *Stat* and *Expr*. We may also have to define separate equations for the various concrete syntax forms these sorts may take on. As an example, consider some of the concrete syntax forms of a C-statement (*Stat*):

```
context-free syntax
  "if" "(" Expr ")" Stat                   -> Stat
  "while" "(" Expr ")" Stat                -> Stat
  "for" "(" Expr ";" Expr ";" Expr")" Stat -> Stat
  "return" Expr                            -> Stat
  {Expr ","}*                              -> Exprs
  getExpr(Stat)                            -> Exprs

variables
  "$Expr"[0-9]*                            -> Expr

equations
  getExpr(if($Expr) $Stat)                 = $Expr
  getExpr(while($Expr) $Stat)              = $Expr
  getExpr(for($Expr1;$Expr2;$Expr3) $Stat) = $Expr1, $Expr2, $Expr3
  getExpr(if($Expr) $Stat)                 = $Expr
```

The point here is that in order to get at the expressions (*Expr*) contained within such statements, we have to write a separate equation for every concrete syntax form. This phenomenon may apply to many sorts on a traversal path[1], so the number of required (boilerplate) traversal equations may become unwieldy.

[1] Expressions themselves in particular have many many operators (+, -, *, /, etc.) which would all require separate equations.

Also, more realistic examples of equations are generally much more complex (and less readable) than the ones above.

In addition, the need to refer to concrete syntax in such equations means the equations will break as soon as a non-compatible modification is made to the concrete syntax. This is especially a nuisance during grammar development, as this process usually involves many changes to the (concrete) syntax, which would necessitate corresponding changes in the equations. It also limits the portability of an ASF specification, since its dependence on the specifics of concrete syntax ties it closely to a syntax specification for a certain language or language dialect. In short, the process requires much boilerplate code, and is both very laborious and error-prone.

Implicit term reriting by traversal functions Traversal functions automate the traversal of parse trees, eliminating the need for writing most of the tedious boilerplate functions. ASF currently offers three types of traversal functions [BKV02]:

1. A *fold* or *accumulator*. This is a type of higher-order function which applies a function f to a data structure s of sort $S1$ (which may be a parse tree) and computes a return value, while leaving s unaltered.

   $f(S_1, S_2, ..., S_n) \rightarrow S_2\ \{traversal(accu, ...)\}$

   The higher-order function is implicit in ASF, and we only have to supply a system of equations for f, for example:

   ```
   context-free syntax
     Id*                    -> Symbols
     f(Program, Symbols)    -> Symbols {traversal(accu, ...)}
     f(FunDecl, Symbols)    -> Symbols {traversal(accu, ...)}
     f(FunCall, Symbols)    -> Symbols {traversal(accu, ...)}
     f(FunDef, Symbols)     -> Symbols {traversal(accu, ...)}

   equations
     f($FunDef, $Symbols)   = getId($FunDecl) $Symbols
     f($FunDecl, $Symbols)  = getId($FunDecl) $Symbols
     f($FunCall, $Symbols)  = getId($FunCall) $Symbols
   ```

   Upon evaluation of these equations, the parse tree is traversed automatically, and the function f is applied to any nodes encountered during this traversal that match the given definition.

2. A *transformer* of the form:

   $f(S_1, ..., S_n) \rightarrow S_1\ \{traversal(trafo, ...)\}$

   This may return a transformed version of its input data structure.

```
fProg(
  int f1(void);
  int f2(void)
  {
    g1();
    f2();
    if(g2(x) < 2) {
      g3((g4));
    }
  }
,
)
```

```
fundecl(f1),
fundef(f2),
funcall(f2,g1),
funcall(f2,f2),
funcall(f2,g2),
funcall(f2,g3),
funcall(f2,g4)
```

Figure 8.1: Input (left) and output (right) of the ASF extractor.

3. An *accumulating transformer*, which returns a (possibly transformed) version of its input data structure of type S_1, as well as a computed value of type S_2:

   $$f(S_1, S_2, ..., S_n) \rightarrow < S_1, S_2 > \{traversal(accu, trafo, ...)\}$$

We can specify directives to guide traversal behaviour (top-down, bottom-up, break, continue) in place of the three periods above.

The implementation of traversal functions used through the (graphical) MetaStudio interface is currently buggy (see section B.1). Furthermore, MetaStudio's 'debug reduce' function does not work. The command line version offers some more verbose output, but it does not list, for instance, which nodes are visited during traversal and which rewrite rules are applied, so its helpfulness in debugging ASF specifications is limited.

Implementation of the extractor Listing 8.1 shows the source code for the function call extractor in ASF. Some example input and output is shown in figure 8.1. In order to enable fact extraction from the command line, I have also developed *suckpt*, described in section 8.2.2, as a separate command line tool. This is convenient for both ad-hoc extractions and more systematic ones (see section 9.2). In addition, its input trees do not necessarily have to be completely well-formed. We can use one extractor to validate the other[2].

8.2.2 Java extractor

Introduction Suckpt, short for suck-subtrees-from-a-syntax-tree, is the Java program that allows one to specify a sort for which it will extract all occurrences from a given syntax tree in a postorder traversal. It spits out the sucked subtrees at the other end as a series of files containing parsetrees in ATerm format, having the specified sort as their root node. See section A.4 for the source code.

Extraction with suckpt is done in several stages. First, we extract all *FunDecl*'s. From those, we extract *Id*'s. We do the same for *FunDef*'s, but from

[2] The Java extractor was originally created because of problems with getting traversal functions to work.

Listing 8.1 Function call extractor in ASF using traversal functions

```
%% Function call extractor
%%
%% extracts:
%%   - function declaration
%%   - function definition
%%   - function calls

equations

[] fProg($FunDecl,$DList) = $DList,fundecl(getId($FunDecl))
[] fProg($FunDef,$DList)
   = fFunDef($FunDef,
             $DList,fundef(getId($FunDef)),
             getId($FunDef))
[] fFunDef($FunCall,$DList,$Id) = $DList,funcall($Id,getId(↙
    $FunCall))


%% the following functions operate on concrete C syntax

[] getId($Cruft? $Type? $Id ( $ParamList ) $FunAttr*;) = ↙
    $Id
[] getId($Id ( $ExprList )) = $Id
[] getId($Cruft? $Type? $Id ($ParamList) $CompoundStat) = ↙
    $Id
```

them we also extract all *FunCalls*. For the source code of suckpt and the batch extractor that invokes in, please consult section A.3. The staging described here is reflected in the ASF extractor of the following section, which is much clearer than the Java/bash extractor. Suckpt is invoked as follows:

```
java SuckPT PARSETREEFILE SORTNAME
```

On the command line, this may look something like:

```
# generate parse table from SDF grammar module
pt-dump -m MyGrammar -o MyGrammar.tbl

# parse a term over its language
# output parse tree in textual ATerm format
sglr -lt  -p MyGrammar.tbl -i MyTerm.trm -o MyTerm.tree.txt

# extract subtrees for sort FunDef
# subtrees are written to to FunDef.tree.txt.[0-9]*
suckPT MyTerm.tree.txt FunDef

# show text of parsed term
unparsePT FunDef.tree.txt.1
```

Since the extracted data are themselves parse trees, they may be manipulated in ASF or SDF again, using the sort by which they were extracted as the start symbol.

The *suckpt* program allows for ad-hoc tree traversals from the command line. We don't need to write an SDF and ASF specification for a traversal function, and it can operate in batch mode.

An application of *suckpt* in the extraction and validation of function calls is discussed in section 9.2. Another application of *suckpt* is may be the following. Parse trees generally are huge; too large to conveniently view, process, print or convert to some visual representation such as a JPEG image. With *suckpt*, we can extract the subtrees we are interested in and operate on them instead of the original huge tree.

Its functionality may be expanded by allowing where-clauses, so one could specify to suck only those trees that are of sort *FunDef* and where, say, the Id of the sucked *FunDef* equals 'foo'. Also, support for some common traversal strategies, such as ASF's top-down, bottom-up and continue, may be implemented.

At the moment, *suckpt* is implemented in such a way that the JVM has to be restarted for every sucking operation. Since extracting function calls involves extracting of *FunDefs*, *FunDecls*, *FunDefs* and *Id*'s, a lot of sucking is going on, and the function call extraction performed on `parser.c` was slow as molasses, but optimizations might be made to remedy this with regards to restarting the tool, disk I/O, not traversing subtrees of impossible ancestors, etc. Then again, we might just use another tool altogether.

Implementation Suckpt performs a complete postorder traversal of a tree. Upon visiting a node, it checks whether that node satisfies certain conditions. If so, the subtree T rooted at that node is extracted. Nodes (ATerms) of the following forms are recognized, while others are ignored:

1. `parsetree(T, ...)`
2. `appl(prod([...], S, ...), T)`
3. `appl(list([...]), T)`

Currently, the only condition implemented is that the context-free sort on the right hand side S of the visited node has the exact name that we specify (e.g. *FunDef*). While this suffices for our purposes, but adding conditions would not be very difficult.

Chapter 9

Validation

9.1 Introduction

The calls extracted using our island grammar should be validated to ensure their correctness, and that of the grammar itself and of the extractor used. The present chapter describes how this is done. As a test case, we will use some of the C source code for the SGLR parser, namely the file `sglr/libsglr/parser.c`.

9.2 Method

Recipe The parsing function $\pi(s)$ maps the string s to its associated parse tree p. Our extraction function $\epsilon_V(p)$ takes all occurrences of the sort V in p to a set of subtrees P which have V at its root node. Our strategy for validating function calls for a given source file f will be as follows:

1. Let s be the concatenation of the text of all files directly or indirectly imported by f; this collects the declarations for all functions callable from f into a single string.
2. Obtain parse tree $p = \pi(s)$.
3. Extract all subtrees p_i from p where the sort at their root node is `FunDecl`, giving $P = \epsilon_{FunDecl}(p)$.
   (a) For every $p_i \in P$, add the function f_i whose declaration it represents to the set D, which represents extracted function declarations and definitions
4. Extract all subtrees p_i from p where the sort at their root node is `FunDef`, giving $P' = \epsilon_{FunDef}(p)$
   (a) For every $p_i \in P'$, add the function f_i whose definition it represents to the set D of extracted function declarations and definitions.
   (b) For every $p_i \in P'$, extract `FunCall`'s, obtaining a set of function calls $Q = \epsilon_{FunCall}(p)$
      i. for every $p_i \in Q$, add the function f_i whose call it represents to the set C of extracted function calls.

5. Compute subset of validated calls $C_{valid} \subseteq C$ (see below)

Note that both C and D are unvalidated. However, it is assumed that they are correct, which is a reasonable assumption since both function declarations and function definitions are easy to recognize. Note that constructs recognized incorrectly as calls may be inadvertently validated in case their Id happens to match a symbol in C or D.

Collecting the declarations First, we need to gather together all the relevant function declarations for the module in question into a single source file. The 'includes' relation, as in source file a includes source file b, is a transitive binary relation. Its transitive closure gives us all the files which may contain the required function declarations and definitions that x can legally call. We can let gcc's[1] preprocessor `cpp` work this out. It traverses the import tree, expanding all occurrences of preprocessor macros, including # include directives. In doing so, we obtain a single 'bundle' file that contains all the function declarations and definitions with a command such as[2]:

```
cpp parser.c | grep -v -e "^#" > foo.c.expanded
```

Information about where the necessary header files are located in the filesystem is generally recorded in a codebase's build system configuration files. Such descriptions are usually rather heterogeneous. I have manually gathered the required paths from the makefile for `libsglr`.

Parsing and fact extraction For our parsing function π, we will of course use the Meta Environment's parser `sglr`. Its use is described in chapter 2. The suckpt extraction approach described in section 8.2.2 provides us with the necessary information to establish the calls relationship which we want to validate.

Validation We will check the extracted calls C against extracted declarations and definitions D. More precisely, we will require the set of valid calls C_{valid} to be as follows:

$$C_{valid} = \{(f_1, f_2) \in C \mid f_1 \in D\} \tag{9.1}$$

Here, f_1 is the calling function and f_2 is the function being called.[3] Expressions such as this one map nicely to the functional programming language *Haskell*. The following Haskell-program gives us the required set C_{valid}:

```
validate :: Eq a => [(a,a)] → [a]  → [(a,a)]
validate calls decls = [ (f1,f2) | (f1,f2) <-  calls, f2 ‘↙
    elem‘ decls ]
```

[1]GNU Compiler Collection, formerly GNU C Compiler.

[2]NB: Function definitions, unlike function declarations, are generally not imported using the #include directive.

[3]Note that this type of validation may eliminate hits that might be considered function calls from a local syntactic viewpoint, even though strictly speaking, C requires them to be declared or defined. Such hits are irrelevant for the purpose of generating call graphs though, since if a function is neither declared nor defined, it can most certainly not be called.

The following Prolog-program gives us the same set:

$$valid(F1, F2) : -declared(F2), calls(F1, F2). \tag{9.2}$$
$$setof(F1/F2, valid(F1, F2), L). \tag{9.3}$$

SQL, like RScript [Kli05] is based on tuple relational calculus, and is therefore a good choice for performing analyses such as these, although more complicated analyses are not so easily performed in it. The following SQL statement, again, gives us the same set:

$$\text{SELECT caller, called FROM calls WHERE called IN} \tag{9.4}$$
$$\text{(SELECT name FROM decls);} \tag{9.5}$$

The extracted information is stored using *sqlite*, an SQL compliant RDBMS that stores the data for a database in a single file. This is a very convenient tool to use from the command line. This storage approach is also highly scalable, since we may use any SQL-compliant RDBMS available. Using such a database as a storage backend can help in performing batch extractions and analyses. SWI-Prolog offers a bridge to SQL called PrologSQL that can translate Prolog predicate calls to SQL statements. GNU Prolog offers a Prolog to C compiler.

Extracting the reference calls relation We extract the reference call graph using *gcc* and a tool called *egypt*. This relies on RTL files produced by gcc to generate a GraphViz dotfile which we can transform into a legible format using the following bash script:

```
gcc -dr SOURCEFILE.c
egypt  --include-external SOURCEFILE.c.00.expand \
  | sed s/[[].*// | sed s/[-]\>/,/g | grep  ","
```

There are several other more common tools for doing things such as these, for example the Unix utility `objdump`. This can show symbols in an object file, or even a disassembly from which we may then extract function calls which, as we argue in section 11.10, is relatively straightforward.

Visualizing the calls I have used GraphViz to produce an example call graph, which is shown in the following section, by means of the following code:

```
imgpath=/home/erik/afstuderen/thesis/images
g=call_graph_actor

rm ${g}.dot
echo "digraph funcalls {" >>${g}.dot
echo "SELECT * FROM calls WHERE called IN (SELECT name FROM↙
     defs);"  \
  | sqlite CALLSDB | sed 's/|/→/' \
  | awk '{printf "%s\n ",$1,$2 }' \
  | grep -i "actor" | sort | uniq >>${g}.dot
echo "}" >>${g}.dot

fmt="dia"
cat ${g}.dot | dot -T${fmt} > ${g}.${fmt}
```

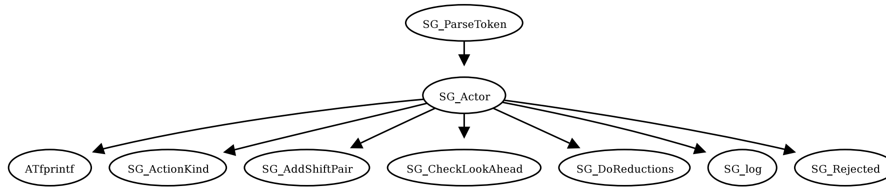


Figure 9.1: Extracted call graph for function `SG_Actor()`

9.3 Results

Since the dataset for `parser.c` would be rather costly in terms of dead trees, I provide a small example dataset and its associated fact base in fig. **??**. The example code is a C rendition of the robot brain example of fig. 1.3. Results of the actual validation can be summarized concisely, since there was disagreement over only two 'functions'; see table 9.1. Figure 9.1 shows an example call/called-by graph for the function `SG_Actor()` in `parser.c`, rendered by GraphViz.

function	**island parser**	**validator**	**egypt/gcc**
`sizeof()`	+	-	-
`get_next_token()`	+	-	-

Table 9.1: Disagreements on positives and negatives

9.4 Discussion

function	**reason**	**grammar modification**
`sizeof()`	is a keyword	treat specially (as keyword)
`get_next_token()`	called via pointer	*Param* may encapsulate *FunDecl*

Table 9.2: Explanation of validation results

Explanations of the results are summarized in table 9.2. Validation filtered out the function calls `sizeof()` and `get_next_token()`. The former is a C builtin which is invoked in a way that is syntactically equivalent to a call to a function of arity 1. The latter is a legal function call by ANSI C. It was filtered out by our validator because no corresponding function declaration or definition was found in the source file or in any of the headers included. In reality however it *was* declared, but through a syntactic construct not completely covered by our island grammar, namely as a function pointer type parameter to `get_next_token()`'s calling functions. This occurs in the following function definition in `parser.c`:

```
token SG_NextToken(int(*get_next_token)(void))
{
    ...
    c = get_next_token();
    ...
}
```

Our syntax definition does define this type of parameter, but it is not injected into the *FunDecl* sort, which explains its omission from the set of declarations and definitions D. It is interesting to note that egypt/gcc also misses this type of function call.

In short, the validator displays no unexpected behaviour (in a syntactic sense), and the island grammar has generated no serious false positives. By crossreferencing the extracted function calls with those extracted by egypt/gcc, we have also established that there were no false negatives, apart from the aforementioned `get_next_token()`. We therefore conclude with great confidence that our island grammar and validator are correct. Any incompleteness may be remedied in a straightforward way with one or a few elementary productions.

Chapter 10

Implementation comparison

10.1 Introduction

This chapter looks at three varieties of the function call grammar from chapter 7

10.1.1 Full Grammar

The first method we will look at is the C grammar that comes with SDF, in particular the version that assumes the C preprocessor has been run on the code. At the time of writing, this grammar had a number of problems that are now reportedly solved:[1]

- The sort *AbstractDeclarator* is not defined by any production.
- Declarations from `stdio.h` cause parse errors.
- Consecutive strings constants ("..." "..."), which occur twice in `parser.c`, generate parse errors.
- Many ambiguities; e.g. in the formal parameters of functions (may be related to *AbstractDeclarator* missing).

10.1.2 Varieties of the function call island grammar

Explicit statements and expressions

This is defined in the function call grammar described in chapter 7. The source code is listed in A.2.

Watery expressions

The following is an excerpt from the watery expression version of `Expr.sdf`, edited for clarity:

[1]It is possible that the results reported below for this grammar are therefore a little inaccurate; the numbers would likely have to be adjusted upwards somewhat, but the points made in the Discussion section stand.

```
lexical syntax
   [0-9]+                              → Digits
   [A-Za-z\_][A-Za-z\_0-9]*            → Id
   [L]? [\'] (([\\]~[])|~[\\\'])+ [\'] → CharConst
   [\^\*\/\%\+\-\!\&\|\~\<\>\.\?\:\=]+ → WATEREXPR {avoid↙
      }

lexical restrictions
   Id          -/- [A-Za-z\_0-9]
   WATEREXPR   -/- [\^\*\/\%\+\-\!\&\|\~\<\>\.\?\:\=]
   Digits      -/- [0-9x]

context-free syntax
%% top-level sorts
   Expr2+                      → Expr
   Expr Expr                   → Expr {reject}
   IslandExpr                  → Expr2 {prefer}
   WaterExpr                   → Expr2 {avoid}

%% water catches operators
   WATEREXPR                   → WaterExpr {avoid}

%% atomic expressions
   CharConst                   → WaterExpr
   "0x"?Digits                 → WaterExpr {prefer}
   Digits?"."Digits("e"Digits)? → WaterExpr
   Id                          → WaterExpr {avoid}
   String                      → WaterExpr {prefer}
   FunCall                     → IslandExpr {prefer}

%% bracket catcher
   "(" Expr ")"                → WaterExpr {avoid}
   "[" Expr "]"                → Subscript
   WaterExpr Subscript         → WaterExpr
```

Watery statements

The following is an excerpt from the watery statement version of `Stat.sdf`, edited for clarity:

```
  "if"                             → Keyword
  "for"                            → Keyword
  "while"                          → Keyword
  "return"                         → Keyword
  ...                              → Keyword

   Keyword                         → WateryStat
   Expr                            → WateryStat {prefer}
%% follow restriction:
   Expr Expr                       → WateryStat {reject}
   "(" Expr ")"                    → WateryStat
%% occurs in for statements:
   "(" Expr? ";" Expr? ";" Expr?")" → WateryStat
   ";"                             → WateryStat
```

```
CompoundStat                          → WateryStat
"{" WateryStat* "}"                   → CompoundStat
```

This specification relies in particular on the list of keywords, which designates the likes of `if`, `for` and `while`, and many others, as such. While the other grammar variants also use this keyword list, their function there is merely to prevent them from being parsed as identifiers; they are not reused there in the actual definition of their corresponding statements.

While this specification is more compact and tolerant than the others, its intent and structure is not as clear as the varieties where statements are explicitly spelled out.

10.2 Comparison

Method The SDF metrics were computed as follows. The number of lines of code (LOC) is computed by filtering out blank and comment-only lines, and counting the number of resulting lines in all the SDF files that constitute the grammar in question. The number of lines of code (LOC) is computed. This is implemented by the following bash script:

```
$cat *.sdf \
  | grep -v -e "^[ ]*\%\%[.]*" |grep -v -e "^[ ]*$" \
  | wc -l
```

The number of productions is determined by the following script, which counts the number of '->' operators:

```
$ cat *.sdf|grep -v -e "^[ ]*\%\%[.]*" \
  |grep -v -e "^[ ]*$"|grep -e "→" \
  |wc -l
```

Counting the number of productions using the tool *dump-productions*, which is part of the ASF+SDF Meta Environment, failed for the full C grammar, also supplied with the meta environment, due to a segmentation fault (which indicates a bug in the software). It did work for my function call island grammar, which reports the number of productions as 709, whereas my own measurement reports 216 productions. This is difference is due to the generation of additional rules in grammar normalization, parse table generation [Vis97], and the expansion of productions using the alternative operator '|' into separate ones. Since my number is taken directly from the written specification, it is more representative of what the grammar engineer deals with. The parser however, uses the productions generated (and properly counted) by *pt-dump*.

The number of priorities, counted as the number of productions duplicated in the priorities sections, are counted by hand.

Results Results have been summarized in table 10.1. Performance comparisons between full grammars, lexical approaches and certain kinds of island grammars are provided in [Lee05].

Discussion It should be stressed that the 'full' grammar is actually incomplete and ambiguous. Therefore, the numbers reported in table 10.1 for the full grammar should probably be higher. Nevertheless, the table clearly shows that

method	**productions**	**priorities**	**LOC**
full grammar	221 + 31*	±50	497
function call island grammar	190	±38	313
idem with watery expr's	116	0	240
idem with watery expr's & stats	100	0	227

*) +31 is a manual correction for the number of productions in Expressions.sdf using the alternative operator instead of the `->` operator.

Figure 10.1: Results

even for nested or complex constructs of interest, considerable gains in terms of number of productions and LOC may be achieved by using an island grammar. However, development was not straightforward. The potential gains depend strongly on the engineer's familiarity with the target language and with island grammar development itself, since it presents a number of challenges not found in the development of full grammars:

1. The Meta-Environment currently does not provide an intuitive and convenient way of resolving island/water ambiguites from within an SDF specification. This problem is described in detail in chapter 12. It has in a number of cases lead to testing many alternative implementations, often richly decorated with various combinations of prefer/avoid attributes, only to conclude that none would work satisfactorily.

2. The definition of certain tolerant constructs, such as the watery expressions, require one to think about these constructs in an unnatural way, that is: a way which runs counter to their nature and, hence, intuition. For expressions, this nature is inherently hierarchic and structured. Attempting to capture such constructs lexically requires a lot of effort to be invested in creative thinking and testing.

   Another example of an unnatural construct definition is provided by the watery statements. There, we have introduced a separation between keywords which require parentheses, such as `if` and `for`), and the actual parenthetic constructs that follow them.

3. Finding island/water sorts and catchers that work correctly can be difficult. This is elaborated on in chapter 13.

For these reasons, island grammar development is difficult for complicated constructs. Accurately assessing whether or not an island approach to a certain problem has is worth the trouble is, consequently, hard.

Chapter 11

Engineering advice

"You can't possibly get a good technology going without an enormous number of failures. It's a universal rule. If you look at bicycles, there were thousands of weird models built and tried before they found the one that really worked. You could never design a bicycle theoretically. Even now, after we've been building them for 100 years, it's very difficult to understand just why a bicycle works - it's even difficult to formulate it as a mathematical problem. But just by trial and error, we found out how to do it, and the error was essential."
— Freeman Dyson in an interview by Stewart Brand (1998)

11.1 Introduction

A number of general notions and patterns have emerged from the experiments in this thesis and from the literature. In this chapter we will explain some of them in a general way against the background of island grammar design. We will also attempt to recast some of the observations into engineering guidelines.

11.2 Signal sequences and catchers

With island grammars, our awareness of the syntactic structure of the source text is generally less than that in the case of full grammars, as much of the structural information is lost in (lexical) water sorts. To compensate for this, we can rely on certain local lexical properties for recognizing constructs. For instance, function calls contain parentheses, so we could look for occurrences of these literals in the source text as indicators of possible function calls. Literals used in such a way are called *signal sequences*. Of course, function calls may not be the only type of construct using parentheses. Looking at expressions, we may also encounter typecasts, strings or bracket expressions containing parentheses:

```
p = (void *) q;                        /* typecast */
char *s = "Donald F. Duck (1943)";  /* string */
n = (1+2);                             /* bracket expression */
```

While signal characters are used to identify islands, the existence of other non-island constructs which also contain these signal characters often necessi-

tates the definition of *catchers*, intended to capture these other constructs and to distinguish them from island constructs. Catchers are generally water constructs; an exception to this is when a catcher may have descendant sorts that are islands. This occurs in bracket expressions for example (see above). We will now look at the design process to see how signal sequences and catchers are used.

11.3 A typical step in the design process

First, we will highlight some typical iteration steps in the island grammar design process by defining an example island construct. As before, we will pick C function calls as our construct of interest, and assume we have an island grammar containing a lexical water sort, creatively named `WATER`, as a starting point.

1. Starting point:

```
lexical syntax
   ~[\ \n\t]+                          -> WATER
```

2. Pick a construct of interest: a function call.

3. Define island(s) for construct of interest:

```
Id "(" {Expr ","}* ")"              -> FunCall
FunCall                             -> Island {prefer}
```

4. Identify signal sequences: opening and closing parentheses `(` and `)`.

5. Exclude these from the water sort:[1]

```
"("                                 -> WATER {reject}
")"                                 -> WATER {reject}
```

6. Define catchers for uninteresting constructs that can now no longer parse because they contain excluded signal sequences:[2]

```
"(" Expr ")"                         -> Expr
STRING                               -> Expr
TypeCast Expr                        -> Expr
"(" Type ")"                         -> TypeCast
[\"][...]*[\"]                       -> STRING
```

This example shows some of the most important recurring design steps and related concepts, which we will use as a basis for discussion in the sections that follow.

[1] This may also be done by removing the characters in question from the lexical definition of WATER.

[2] Moonen defines catchers (which he does not name as such) as *LAYOUT*. [Moo01] The advantage of this is ease of specification. The disadvantages is that such cachers are used in a context-unaware manner.

11.4 Tolerance

Island grammars can provide increased tolerance. This tolerance is achieved through water(y) sorts, which are generally more liberal than their elaborately specified full grammar counterparts. Tolerance can be a desirable property of a grammar, since it facilitates parsing of language dialects and less well-formed source code. In addition, a tolerant lexical specification may reduce the number of productions that need to be specified. As an example, we could capture C-keywords such as `if`, `while` and `for` with a tolerant lexical definition, instead of defining every keyword explicitly:

```
lexical syntax
[a-z]+ -> WaterKeyword
```

11.5 Correctness

Consider the following grammar:

```
lexical syntax
  ~[]*              -> Water

context-free syntax
  Water             -> Program
```

This defines an island grammar, namely a maximally tolerant one, in which the only water sort is as liberally defined as possible, and a *Program* consists of only water, without islands.[3] Suppose some construct(s) of interest were given. With respect to these, the grammar generates only false negatives, since no islands representing them have been defined.

The desirability of tolerance notwithstanding, a grammar and its constituent water sorts should obviously not be so tolerant as to admit false negatives, which would cause the grammar to be incorrect. On the other hand, excessively tolerant island sorts may cause false positives. This shows that there is a there is a tradeoff between tolerance and correctness.

Note that false negatives should be avoided at all cost, since they lead to irrecoverable information loss during parsing. False positives on the other hand may be filtered out later using semantic analysis; for an example of this please see chapter 9. It is generally a good idea to prefer possible parse errors over possible false negatives, since the latter may go unnoticed, while the former may be remedied with a refinement of the grammar used, as shown in section 11.3. This section also demonstrates how signal sequences can be used to ensure that certain constructs which contain them will not accidentally be parsed as water, thus reducing the risk of false negatives.

While in many cases we may be fairly confident that an island grammar performs as expected given careful design, it is generally not possibly to infer correctness from the grammar, since it may generate an infinite language, and there is always the possibility of false negatives. Section 15.2 presents a testing framework for SDF that should help ensure correctness.

[3]While island grammars are generally defined as grammars that actually include island sorts, we deviate from this here in order to make a point.

11.6 Compositionality

The definition of watery sorts affects compositionality. Consider for example the following watery definition for expressions:

```
lexical syntax
  ~[]*                    -> Expr %% capture every character
```

While this production will certainly capture all expressions in the target language, it would likely conflict with other sorts in the resulting composite language if we were to merge the production into another specification (i.e. composition). For instance, if our example *Expr* sort were used in the following definition, it would swallow the neighbouring parentheses, if-statement and *Stat*:

```
context-free syntax
  "if" "(" Expr ")" Stat   -> Stat
```

This illustrates how liberal water sorts tend to *overflow* their intended boundaries, causing unintended parses or ambiguities. So, liberal watery sorts can reduce compositionality. Hence, from a compositionality and correctness standpoint, it is desirable to specify such a sort with as little lexical freedom as possible. Note the tradeoff between tolerance and liberal specifications on one hand, and compositionality on the other.

We can limit the possible detrimental effects of water sorts on compositionality by creating specific water sorts for separate constructs. For instance, expressions have their own water sort (e.g. *ExprWater*) and so do statements (*StatWater*). This allows us to tailor their lexical definitions exactly to the constructs that need to be captured by the water sort. (This point was left out of the example of section 11.3 for simplicity.)

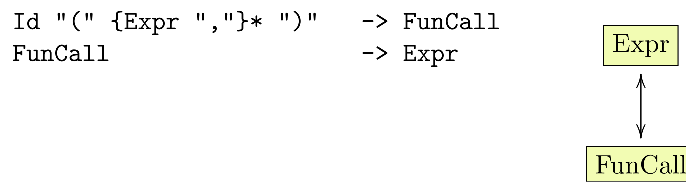
Note that the concept of compositionality is used here in the sense of a *degree*, as opposed to a discrete yes/no matter. Summarizing, we can say that while island grammars are in general not compositional, the *degree* to which this is the case may be reduced through the restrictive (as opposed to liberal) specification of watery sorts.

11.7 Nested sorts

Nesting occurs when sort A can occur as a subterm in sort B. This is the case with function calls and expressions in our function call grammar:

```
y = 3 * f(g(x)+1)
```

Here, the top level sort is an (assignment) expression, which is not in itself a construct of interest. Within the expression however, we encounter a function call to `f()` of the form `f(g(x)+1)`. Within this function call, another expression occurs, namely `g(x)+1`, which in turn contains another function call: `g(x)`. So function calls are nested within expressions, and vice versa. For this reason, both expressions and function calls are also nested recursively within themselves. In the syntax defintion, we can see that both sorts occur in the left hand side of each other's productions:



Since function calls, and not expressions, are our constructs of interest, we would like to define them with as little effort as possible. It would seem that the obvious answer to this is to create a water sort for expressions, allowing us to capture them lexically if possible.

However in doing so, we must realize that while expressions are not constructs of primary interest, they are significant in that they may *contain* constructs of primary interest. Expressions, as such, are constructs of *secondary* interest. In their definition we must account for this fact, and make sure that function calls stand out from the watery remainder of the expression sort. We can do this by looking for the signal sequences that a function call may contain, as we have illustrated in a previous paragraph, and excluding them from the water.

Expressions are nested within other sorts, from which we must be able to distinguish them, in order to identify them in the context of an entire C program.

This means we will have to devise water sorts and catchers for each nesting ancestor sort as well. All in all, the required effort is considerable. Chapter 13 describes a method to investigate some of the possible conflicts that may occur as a result of nesting.

11.8 Shortcuts

Island grammars are supposedly ‘lightweight’, meaning that they are considered to require less specification effort. Section 7.4 describes how the production for function calls was recycled to capture function attributes, which occur as part of function declarations. As it happens, this was rather quickly found to work satisfactorily, which would seem to support the idea of lightweightness associated with island grammars.

The paragraph on statement separation in section **??** on the other hand, describes a shortcut that allows us to parse multiple statement types with a single production. (It seems fair to qualify certain shortcuts as ‘hacks’. While the notion of a hack might seem to run contrary to the general idea of science, it should be considered an integral part of island grammar development, as we take sometimes unconventional shortcuts around having to define a full grammar. At the very least, one might wonder which shortcuts are hacks, and which are not.)

Philosophy aside, this particular shortcut resulted in a reduction in the total number of required productions which, again, seems like a testimony to the lightweight nature of the island grammar approach. However, while its definition may seem obvious, it took a substantial amount of time to come up with and test; especially since this was done in the initial experimentation phase of this

research project.

Creativity, as opposed to verifiable methodology, plays a role of significance in devising shortcuts. Moreover, there is no guarantee upfront that some shortcut we may come up with will actually work. Sometimes, inadequacies may only come to light in a late stage of development, which is generally very costly in software development. While the trial and error approach is more or less inherent to grammar engineering, island grammars add a new dimension of unpredictability in this respect. This makes the costs and benefits difficult to estimate, and the approach risky.

11.9 Island/water ambiguities

A key issue in island grammar development is the frequent occurence of island/water ambiguities. Suppose we had a grammar that correctly recognized some construct of interest as an island. This does not rule out the possibility of the construct being recognized as water too, at the same time.

Figure 11.1 shows an example of how a function call `foo(x)` might find itself in such a predicament; it is correctly identified as an island, but it is also valid water because the definition of water allows for an *Id* immediately followed by a *BracketExpr*.

The problem is that the function call argument `x` surrounded by parentheses is mistaken for a bracket expression. Indeed, the parser cannot be expected to distinguish between these, as that would require more awareness of the syntactic structure than the use of a water sort permits, in this case. The solution to this very common type of problem would be to prefer our island construct over the water construct. Chapter 12 looks at which disambiguation methods we might use to this effect.

11.10 Portability across languages

Some languages are inherently more difficult to parse than others. A notoriously difficult language in this respect is C++. It offers many features that complicate its syntax, such as namespaces, templates, multiple inheritance, operator overloading, and polymorphism. The following quote, if to nothing else, certainly applies to its syntax:

> *"If C gives you enough rope to hang yourself, then C++ gives you enough rope to bind and gag your neighborhood, rig the sails on a small ship, and still have enough rope to hang yourself from the yardarm"* — Anonymous, The UNIX-HATERS Handbook

Fortunately, some languages are easily parsed. We will now take a look at some of the properties that affect this 'parseability'.

With island grammars, the precise syntactic structure of the environment in which a given construct of interest is embedded is often not known. Because of this, the recognition of constructs relies heavily on local lexical features. Therefore, languages which make this local recognition of constructs of interest easy through their (lexical) syntax, are more suitable candidates for island grammar descriptions.

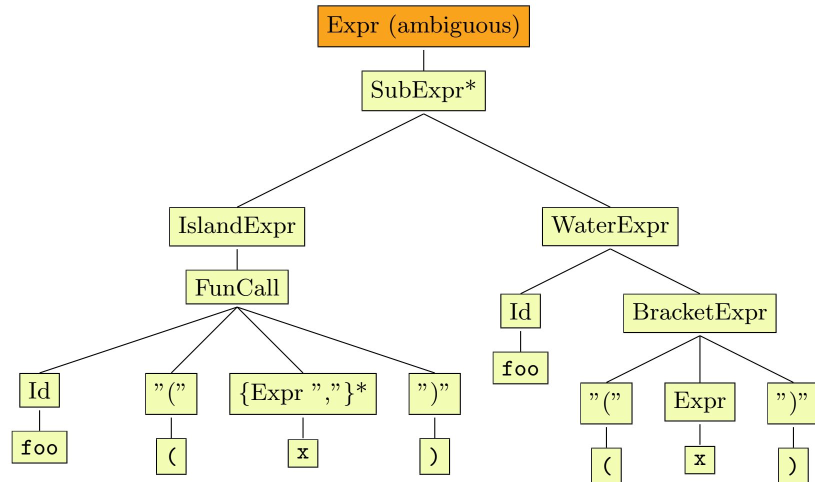


Figure 11.1: Typical island/water disambiguation problem in the form of an expression `foo(x)`. The desired subtree is the one on the left in which `foo(x)` is parsed as a *FunCall*.

```
// Hello World in PHP

f('Hello World!');

function f($msg)
{
  print($msg);
}
```

```
% Hello World in Matlab

f('Hello World!');

function rv = f(msg)
  disp('msg');
```

Figure 11.2: Function definitions signaled by the signal sequence '`function`'

Some languages offer clear lexical cues to demarkate certain constructs. For example, Python, JavaScript, PHP and Matlab require the keyword `function` to indicate that what follows is a function definition (fig. 11.2). Assembly languages feature mnemonic instructions which can serve as signal sequences. Example listing 1.4 shows how a procedure call in CMOS/NMOS 6502 assembly is preceded by a `JSR` instruction. The Intel x86 family of processors has the `call` instruction for procedure calls. MS-DOS and Linux/x86 based systems implement system calls as interrupts, generated by the `int` instruction. PowerPC, a RISC architecture, provides instructions such as `bl` (branch with link) and `sc` (system call).

Assembly languages typically also contain little or no nested constructs, but instead are basically a long list of single-word mnemonics followed by a comma-separated list of operands, with the occasional label thrown in for good measure. Expressions can occur, but they may only contain constant subexpressions that can be evaluated during preprocessing or compilation, such as `2*(my_constant+3)`, so tasks like function call extraction are quite straightforward compared to the C case.

source language	syntactic idiosyncracies
XML, HTML	-
assembly	very regular flat structure
	mnemonics are good signal sequences
C	
C++	operator overloading
Python, Haskell	off-side rule[4]
COBOL	comment columns at fixed positions
FORTRAN	identifiers may contain layout (req's lookahead)
	features `CALL` keyword
PL/I	keywords allowed as identifiers
Matlab/Octave	function calls with and without ()
	allows undeclared library calls
	array index and function call both use ()
	function definitions may be nested
...	dynamic type inference, polymorphism

Table 11.1: Language features

Some languages such as assembly code are inherently easy to analyze due to their flat syntactic structure and easily recognizable signal constructs (the mnemonics). Nesting of syntactic constructs - a problem begging for syntactic analysis - occurs very often in XML-only or HTML-only documents. They do not generally adhere to their respective formal language specifications very strictly, which requires tolerant parsers, which in turn may be implemented using island grammars, as we have seen. Their easily recognizable signal constructs (tags of the form `<foo ...>`, `</foo >` or `<foo />`), again, facilitate their implementation. In addition, it can be much more convenient, and faster, to implement a syntactic analysis in a couple of island grammar rules than to do so by importing an entire XML specification and defining constructs of interest on top of that.

There are many other specification languages that are essentially lists of name/value pairs; *GNU autotools* files, UNIX config files, address books, etc. Defining an island grammar for these is quite easy, and allows us to abstract away from their concrete syntax, which in turn enables analysis and semantification through generic methods.

Of course, certain languages have features which make them especially *unsuitable* for any but the most elementary island grammar analyses. For example, Matlab uses the same notation to denote array indexing and function call parameters:

```
y = 3 * f(x)       % is f a function or an array?
```

Fortran allows spaces inside of identifiers [Aho86]:

```
DO 5 I = 1.25      ! DO5I is an identifier
DO 5 I = 1,25      ! "DO" Label Id "=" Const "," Const
```

These languages not only require significant awareness of syntactic structure for many tasks. The Fortran exampe for instance requires a lookahead of 5 characters, which is not a problem for SDF/SGLR, but can be for other common parser types such as LALR(1) class parsers[5], to which the ubiqitous Yacc belongs. To get around this, some Fortran parsers have two stages; one recognizes the problematic constructs requiring a lot of lookahead lexically, while a subsequent stage performs further analysis on the lexed constructs. This is essentially an island grammar approach.

Certain problems such as identifier (type) recognition (and hence function call recognition) are undecidable by *any* context-free parsing without semantic analysis. This is the case with Matlab, for example.

11.11 When to use island grammars

The result of parsing some (program) text over a grammar is a syntax tree, which decribes the internal and external syntactic structure of the constructs recognized by the grammar. Lexical analysis on the other hand can identify substrings in a text and point out the positions in which they occur (just as parsing can), but provides no information about the structural context of the recognized constructs. The patterns which these substrings must match are specified in the form of regular expressions. If some structural awareness is desired, it has to be explicitly programmed around or into the actual lexing component(s).[6] Definitions for lexers can also get rather messy, as we have seen in the case of Doxygen/Lex in section 5.5.

In case the problem we are trying to solve needs some awareness of the internal or external syntactic structure of a construct of interest, we should consider using a grammatical approach (i.e. parsing). Though lexical analysis is generally faster for simpler tasks, island grammars offer performance benefits with respect to full grammars [Lee05]. Island grammars are particularly well-suited int the following areas:

[5]The 1 in LALR(1) indicates a lookahead of one character, while Fortran requires unbounded lookahead, which SDF/SGLR provide.

[6]Doing so takes lexical analysis in the direction of parsing, which raises the question of whether one shouldn't be using a parser to begin with.

1. Tolerant parsing
2. Identifying high level constructs (which have little or no nesting)
3. Identifying constructs containing characteristic signal sequences

Example applications of this are language cocktails that occur in common web scripting languages, such as HTML mixed with JavaScript, JSP, PHP or ASP [SCD03] or COBOL with embedded CICS [KL03]. Class hierarchies are also typically high-level constructs, for example in Java, C++, Python, Ruby and C++. Comments are another example. They are especially easy to recognize due to their signal characters such as '//' in Java, although matching them up with constructs they describe is much more involved.

- When parsing a certain language that is inherently suitable for island parsing due to its flat structure or characteristic signal sequences. Assembly language is an example of this (see section 11.10).
- In the absence of a full grammar; an ad hoc grammar approach may be preferable over lexing. [Moo02]
- In conjunction with a full grammar. A tolerant grammar (which may be considered an island grammar) can be semi-automatically derived from this. This offers performance benefits compared to full grammars, and quality benefits compared to lexing. [KL03]
- As an agile prototyping tool in the course of developing a full grammar. Water sorts can be used initially to cover simple examples so other parts of the grammar can be tested. Later, the water sorts can be refined into more detailed productions.
- For identifying simple constructs in *multiple* languages with a *single* grammar. For example, an imports-grammar that recognizes C's #include, Java's `import`, Python's `import`, PHP's `include()` and `require()`, etc.

11.12 Miscellaneous points of advice

- Aggregate occurrences of consecutive water characters into single parse tree node as much as possible to prevent excessive parse tree sizes and reduce the usage of system resources.
- Create separate water types for every island sort to isolate the possibly adverse effects of the water on the grammar's compositionality as best as possible.
- If speed and memory constraints are of little importance, allowing for ambiguities in the grammar may be convenient from a rapid prototyping perspective. Ambiguities may be resolved in a post-parsing stage by a tool such as *filterPT*. Note however that their regular occurrence may have a severe negative impact on parsing performance.
- Ambiguities do, however, indicate vaguenesses in our specification, which should alert us to the possibility of false negatives.

- If expressions are not constructs of interest, then expression grammars may be good candidates for a water treatment. Usually, they are quite voluminous due to the large number of productions and priority declarations involved. A lot of specification, testing and debugging may be avoided by capturing them lexically as exemplified in section **??**. (Alternatively, since they are also quite uniform across many programming languages, copying them from an existing grammar and editing them into shape may be preferable.)

- It *almost* goes without saying that the grammar itself should be documented well. For island grammars, which may employ several hacks, tricks, shortcuts and whatnot, this is especially necessary.

- Consider using tools such as *SdfMetz* [AV07] to help optimize the grammar if necessary; eliminate redundant injections, etc.

Chapter 12

Disambiguation

12.1 Introduction

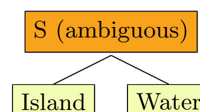
The watery expression grammar from chapter 10 reveals some ambiguity problems that are very common with island grammars, and which cannot easily or reliably be solved using disambiguation techniques currently available in the Meta-Environment. In this chapter, we will elaborate on these problems and attempt to present a solution.

12.2 Existing disambiguation mechanisms

12.2.1 Prefer/avoid

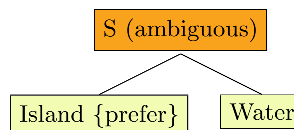
Consider a simple grammar and an example parse tree of some ambiguous term over this grammar:

```
context-free syntax
  Island -> S
  Water  -> S
  ...    -> Island
  ...    -> Water
```



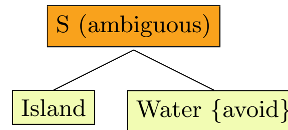
This ambiguity can be resolved in SDF by tagging *Island* with a `prefer` attribute:

```
context-free syntax
  Island -> S
  Water  -> S
  ...    -> Island {prefer}
  ...    -> Water
```

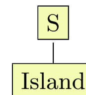


Alternatively, we might tag *Water* with `avoid`, which is the inverse of the `prefer` attribute:

```
context-free syntax
  Island -> S
  Water  -> S
  ...    -> Island
  ...    -> Water {avoid}
```



In our example, both have the same effect, which is the elimination of the unwanted *Water* subtree on the right, resulting in the following unambiguous tree:



Now consider the ambiguity in figure 11.1, in which a function call occurs as part of an expression sort, which is basically a list of subexpressions. Due to its definition, this expression sort may be ambiguous in the presence of function calls, since these may be interpreted as either a function call (the desired interpretation) or an identifier (*Id*) followed by a bracket expression (*BracketExpr*). Given the prefer/avoid construct, intuition might suggest tagging the productions for *IslandExpr* and *WaterExpr* with prefer and avoid, respectively:

```
context-free syntax
  IslandExpr     -> SubExpr {prefer}
  WaterExpr      -> SubExpr {avoid}
```

However, looking at the tree of figure 11.1, we can see that the nodes for *IslandExpr* and *WaterExpr* do not occur directly under the ambiguous *Expr* node. Unfortunately, the basic prefer/avoid can only disambiguate situations in which the ambiguous nodes *do* occur directly underneath an ambiguous node. SGLR does have mechanisms in place to disambiguate using prefer/avoid by descending into subtrees, namely:

- Prefer/avoid counting: subtrees with more prefers and less avoids are favoured over subtrees with less prefers and more avoids.
- Injection counting: trees with less injections are favoured over trees with more injections.

Unfortunately, the use of these heuristics is discouraged, and they have been disabled in MetaStudio for the following reasons:

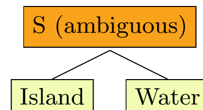
> "[...] these disambiguations silently disambiguate and therefore hide important problems of a syntax definition. They might also disambiguate in a 'context-sensitive' manner; choosing one alternative in some context, and another in a different context. Several examples of these effects have lead to the decision of turning them off."[1]

So, these mechanisms can lead to unpredictable and unwanted results. Therefore we will not rely on them for disambiguating the type of problem described above.

[1] Source: release notes of the Meta-Environment 1.5.

12.2.2 FilterPT

An alternative to the prefer/avoid approach is the command line tool *FilterPT*, the function of which is to minimize or maximize the occurrence of certain sorts in a parse forest. [Lee05] Suppose that we had a parse forest with the following ambiguous subtree in it, and that we wanted to keep the *Island* subtree and discard the *Water* subtree:



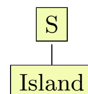
We could disambiguate this example with FilterPT from the command line as follows:

```
filterpt -N Island
```

The switch `-N` tells FilterPT to maximize the occurrence of the sort *Island*. We may also use FilterPT in the converse way, by *minimizing* a given sort, indicated by the switch `-n`:

```
filterpt -n Water
```

In our example, both approaches would result in the unambiguous subtree:



Contrary to the basic prefer/avoid mechanism, this filter does take the whole subtrees into consideration, by counting the occurrences of the given sort in them and minimizing or maximizing accordingly. This means we could disambiguate the parse forest of figure 11.1 by maximizing the sort *FunCall*. This would eliminate the right subtree in favour of the left one, which is what we intended to do.

However, while our intention is to disambiguate based only on the topmost occurrence of *FunCall*, FilterPT also descends into the subtrees *underneath* this node. In doing so, it takes subtrees into account that have no relevance whatsoever to the actual disambiguation problem. This may also affect the disambiguation in unpredictable ways. Consequently, we cannot be sure that FilterPT will not eliminate subtrees that we would have liked to keep.

Moreover, the subtrees may themselves contain water/island ambiguities, so the order in which FilterPT is applied in case of multiple disambiguations based on different sorts becomes important. Determining the right order would require a detailed analysis of possible occurrences of nested ambiguous trees, if we want to make sure we are not discarding important subtrees unwittingly.

12.2.3 Priorities

Priorities prevent certain productions from occurring as direct children of other productions. Consider the following example grammar (unnecessary details left out):

```
lexical syntax
  [0-9]*        -> Expr
context-free syntax
  Expr "*" Expr -> Expr
  Expr "+" Expr -> Expr
```

Now suppose we were to parse the term `3*a+b` over *Expr*. This term may be interpreted semantically to represent the arithmetic expression $(3a) + b$. As we defined it however, it may be parsed in two ways and is therefore ambiguous:

Using priorities, we can make our parser mimic the familiar arithmetical operator binding rules, which say that multiplication binds more strongly than addition. This eliminates the possibility of the tree on the right, which represents an incorrect parse corresponding to $3(a + b)$, occurring in our parse tree. We can achieve this by adding a priorities section to our SDF specification:

```
context-free priorities
  Expr "*" Expr -> Expr >
  Expr "+" Expr -> Expr
```

This expresses the restriction that (the production rule for) '`+`' cannot occur as a direct child of '`*`' in the parse tree, as is the case in the tree on the right. This type of disabiguation operates on ambiguous subtrees whose nodes are the same, but arranged in a different order. It is intended to disambiguate expression grammars, and cannot be used to resolve our example ambiguity.

12.3 Discussion

The existing disambiguation methods discussed above are either wholly unable to disambiguate the common type of island/water ambiguity illustrated in figure 11.1, or cannot do so practically (e.g. through invocation from an SDF specification) or predictably:

method	**applica-bility**	**predic-tability**	**in SDF**	**in MetaStudio**
prefer/avoid	very limited	?	yes	yes
prefer/avoid counting	very limited	no	n/a	yes
injection counting	very limited	no	n/a	yes
priorities	no	n/a	yes	yes
reject	limited	n/a	yes	yes
FilterPT	yes	limited	no	no

Table 12.1: Overview of current disambiguation methods

In the following section we will propose a new type of disambiguation construct that should solve these problems, if it is correctly implemented and, importantly, can be invoked from SDF.

12.3.1 A prefer-to filter

Using injections, we can create a situation in which the ambiguity can be resolved by looking at a certain *level* in an ambiguous subtree. Our familiar figure 11.1 tree demonstrates this: the sort *FunCall* is first injected into *IslandExpr*, while all the expression subtypes other than function call are injected into *WaterExpr*.

```
context-free syntax
  IslandExpr      -> SubExpr {prefer-to(WaterExpr)}
  WaterExpr       -> SubExpr

  FunCall         -> IslandExpr
  ...             -> WaterExpr
```

A prefer-to filter would descend ambiguous subtrees level by level in a lock-step manner, checking all subtrees at the same depth and progressing downward from the their root nodes. Upon encountering a construct marked with `prefer-to(S)`, it would discard all subtrees with a production `... -> S` at that level. Clearlym, this strategy would work in our example case. Moreover, the technique of introducing injections and using them to mark (subtrees as) islands or water at the same level can be applied generically. It is especially useful in tackling ambiguities occuring 'under' list constructors in a parse tree, which is a problem for the avoid/prefer construct.

A filter such as the one described here may be implemented in ASF; an explanation and examples of post-parse disambiguation filters can be found in [BKV02]. The additional attributes required can be specified in SDF, although the automatic invocation by SGLR would require modifications to the parse table generator and parser/parser generator. (A basic data flow diagram for these components is depicted in figure 14.2.)

Chapter 13

Lexical inheritance analysis

13.1 Introduction

Consider a language modeled in part by the example grammar in figure 13.1, and represented visually in 13.2.[1] Let's say *FunCall* is our sole construct of interest. Since we would like to specify the uninteresting constructs in the form of water productions, and water is, in principle, defined lexically, it follows that we must to some extent rely on lexical properties to distinguish between water and non-water. We can use signal sequences to do this.

```
lexical syntax
  [\"][A-Za-z0-9\(\)\,]*[\"] -> String     %% production p_1
  [A-Za-z0-9]+               -> Id         %% production p_2

context-free syntax
  Id "(" {Expr ","}* ")"     -> FunCall    %% production p_3
  String                     -> Expr       %% production p_4
  FunCall                    -> Expr       %% production p_5
```

Figure 13.1: Mini example grammar for C expressions and function call

Function calls contain parentheses (and possibly commas) so we might use those as signal sequences in spotting function calls. However, looking at the figures, we see that strings may also contain these characters. This could lead to conflicts resulting in ambiguities, false positives or parse errors.

If we are to construct a correct island grammar that does not exhibit such problems, we must have some systematic method for detecting conflicts between literals of the various sorts. The following sections propose one such method, which should serve to inform decisions about whether or not a candidate set of productions is suitable for inclusion in the grammar being designed.

We begin by introducing the $\widehat{\Delta}^*$ operator, which we can use to construct

[1]The lexical definition for the sort *String* would normally include many more characters. This is typically specified by listing the characters that are *not* part of the sort and then taking their complement, but this was thought to be a slightly more enlightening considering the subject we are trying to introduce.

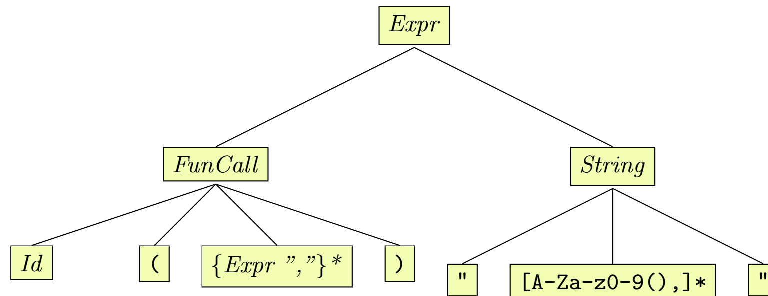


Figure 13.2: Visual representation of example grammar from fig. 13.1

sets of inherited sorts. For instance, with reference to figure 13.2, we can say that the *Expr* sort *inherits Id* (in this case through *FunCall*).

We then introduce the $\mathcal{L}$ operator, whose definition relies on that of the former. It gives us the literals used in or inherited by some production or set of productions, as well as all their substrings. Using these operators, we can determine of any of these substrings are in conflict.

13.2 Definitions

Consider a context-free grammar $G = (V, \Sigma, P, \widehat{S})$ as defined in section 1.1. The variables s, s', s_i represent sorts in $V \cup \lambda$, where λ is the empty sort. The variables p, p', p_i range over the productions P, and t_i are terminal symbols, i.e. strings over the alphabet Σ. A production p can be written as $p : s_0 \rightarrow s_1 t_1 ... s_n t_n$. We will write $p : s_0 \rightarrow ...s'...$ to denote that production p contains the sort s' in its right hand side, which makes s' a child of sort s_0 and, loosely speaking, of production p. We can now define two important functions:

- The **child productions** operator Δ, which gives us all the productions of sorts that are *directly* referenced by the subset P' of the productions P. It has the form $\Delta : P^m \rightarrow P^n$ where $m, n \in \mathbb{N}^+$. Its definition is:

$$\Delta(P_{s'}) \equiv \{p \in P' \mid p : s_0 \rightarrow ...s'...\} \tag{13.1}$$

- The **descendant productions** operator $\widehat{\Delta}^*$, which gives us all the productions of sorts that are *directly or indirectly* referenced by $P' \subseteq P$, modulo cyclic productions. It has the form $\widehat{\Delta}^* : P^m \times V^k \rightarrow P^n$ where $k, m, n \in \mathbb{N}^+$, and is defined as:

$$\widehat{\Delta}^*(P_s) \equiv P_s \cup \widehat{\Delta}^*(\Delta(P_s), \{s\}) \tag{13.2}$$

The latter function serves to call the recursive form of $\widehat{\Delta}^*$:

$$\widehat{\Delta}^*(P_s) \equiv P_s \cup \widehat{\Delta}^*(\Delta(P_s), \{s\}) \tag{13.3}$$

$$\widehat{\Delta}^*(P_s, V') \equiv \begin{cases} \emptyset & s \in V' \vee P_s = \emptyset \\ P_s \cup \widehat{\Delta}^*(\Delta(P_s), V' \cup \{s\}) & \text{otherwise} \end{cases} \tag{13.4}$$

The following property applies, which we will use as a reduction step in our equations:

$$\widehat{\Delta}^*(P_{s_1} \cup P_{s2}, V') \equiv \widehat{\Delta}^*(P_{s_1}, V') \cup \widehat{\Delta}^*(P_{s_2}, V') \tag{13.5}$$

Now we shall introduce the $\mathcal{L}$ operator. Suppose P' is a set of productions, which we might have found using the operators just defined. We will denote the regular language defined by taking all the substrings of all literals occuring in P' by $\mathcal{L}(P')$.

$$\mathcal{L}(\{s_1 t_1 ... s_n t_n\}) = \{t_1 ..., t_n\} \tag{13.6}$$

$$\mathcal{L}(p_1 \cup p_2) = \mathcal{L}(p_1) \cup \mathcal{L}(p_2) \tag{13.7}$$

The terminal symbols t_i all define regular languages, sinces they are expressed in terms of regular expressions (in SDF's `lexical syntax` sections) or as string literals (in SDF's `context-free syntax`). For a singleton set of productions of the form $p : s_0 \rightarrow s_1 t_1 ... s_n t_n$, the set $\mathcal{L}(\{p\}) = \{t_1 ..., t_n\}$ is certainly a regular language, namely $t_1 \cup ... \cup t_n$. This is because the set of regular languages is closed under union. The same argument applies to equation 13.7. Hence, the operator $\mathcal{L}$ defines a regular language.

Since we are going to be looking at substrings, we provide their definition here for the sake of qcompleteness:

Definition A **substring** of a string $w = w_1 w_2 ... w_n$ is a string $w_i w_{i+1} ... w_j$ where $1 \leq i \leq j$ and $i \leq j \leq n$. For example, `bo`, `w`, `rain`, and `rainbow` are all substrings of the string `rainbow`.

13.3 Example

As an example, we compute the set $\widehat{\Delta}^*(P_{String})$:

$$\begin{aligned} \widehat{\Delta}^*(P_{String}) &= \widehat{\Delta}^*(P_{String}, \emptyset) \\ &= P_{String} \cup \widehat{\Delta}^*(\Delta(P_{String}), \emptyset) \\ &= P_{String} \cup \widehat{\Delta}^*(\emptyset, \emptyset) \\ &= P_{String} \cup \emptyset \\ &= P_{String} \\ &= \{p_1\} \end{aligned}$$

This tells us that *String* has no descendants, other than possibly *Id* itself, which is evidently true from looking at the grammar. And now for *Expr* (*FunCall* is abbreviated to *FC*):

$$
\begin{aligned}
\widehat{\Delta}^*(P_{Expr}) =\ & P_{Expr} \cup \widehat{\Delta}^*(\Delta(P_{Expr}), \{Expr\}) \\
=\ & P_{Expr} \cup \widehat{\Delta}^*(P_{String} \cup P_{FC}, \{Expr\}) \\
=\ & P_{Expr} \cup \widehat{\Delta}^*(P_{String}, \{Expr\}) \cup \widehat{\Delta}^*(P_{FC}, \{Expr\}) \\
=\ & P_{Expr} \\
& \cup P_{String} \cup \widehat{\Delta}^*(\Delta(P_{String}), \{String, Expr\}) \\
& \cup P_{FC} \cup \widehat{\Delta}^*(\Delta(P_{FC}), \{FC, Expr\}) \\
=\ & P_{Expr} \\
& \cup P_{String} \cup \widehat{\Delta}^*(\emptyset, \{String, Expr\}) \\
& \cup P_{FC} \cup \widehat{\Delta}^*(P_{Id} \cup P_{FC}, \{FC, Expr\}) \\
=\ & P_{Expr} \\
& \cup P_{String} \cup \emptyset \\
& \cup P_{FC} \cup [\widehat{\Delta}^*(P_{Id}, \{FC, Expr\}) \cup \widehat{\Delta}^*(P_{FC}, \{FC, Expr\})] \\
=\ & P_{Expr} \cup P_{String} \cup P_{FC} \\
& \cup [(P_{Id} \cup \widehat{\Delta}^*(\Delta(P_{Id}), \{Id, FC, Expr\})) \cup \emptyset] \\
=\ & P_{Expr} \cup P_{String} \cup P_{FC} \\
& \cup [P_{Id} \cup \widehat{\Delta}^*(\emptyset, \{Id, FC, Expr\})] \\
=\ & P_{Expr} \cup P_{String} \cup P_{FC} \cup [P_{Id} \cup \emptyset] \\
=\ & P_{Expr} \cup P_{String} \cup P_{FC} \cup P_{Id} \\
=\ & \{p_4, p_5, p_1, p_3, p_2\}
\end{aligned}
$$

Now we determine which literals occur in certain productions:

$$
\begin{aligned}
\mathcal{L}(P_{FC}) &= \mathcal{L}(\{p_3\}) \\
&= \{\texttt{[\textbackslash(]},\ \texttt{[\textbackslash)]},\ \texttt{[\textbackslash,]}\}
\end{aligned}
$$

Using the results obtained above, we compute the following set:

$$
\begin{aligned}
\mathcal{L}(\widehat{\Delta}^*(P_{Expr}) \setminus P_{FC}) =\ & \mathcal{L}((P_{Expr} \cup P_{String} \cup P_{FC} \cup P_{Id}) \setminus P_{FC}) \\
=\ & \mathcal{L}(\{p_4, p_5, p_1, p_3, p_2\} \setminus \{p_3\}) \\
\supseteq\ & \mathcal{L}(\{p_1\}) \\
=\ & \{\texttt{["][A-Za-z0-9\textbackslash(\textbackslash)\textbackslash,]*["]}\} \\
\supseteq\ & \{\texttt{[\textbackslash(\textbackslash)\textbackslash,]*}\}
\end{aligned}
$$

Now we look for common substrings of literals between the sorts *FunCall* and *Expr*:

$$\begin{aligned} & \mathcal{L}(P_{FC}) \cap \mathcal{L}(\widehat{\Delta}^*(P_{Expr}) \setminus P_{FC}) \\ \supseteq\ & \{\texttt{[\textbackslash(]},\ \texttt{[\textbackslash)]},\ \texttt{[\textbackslash,]}\} \cap \{\texttt{[\textbackslash(\textbackslash)\textbackslash,]*}\} \\ \neq\ & \emptyset \end{aligned}$$

13.4 Results

We will consider the outcomes of our example analysis as our dataset. The productions associated with *FunCall*'s descendant sorts are listed in table 13.1.

p_i	V	$\mathcal{L}(\{p_i\})$	conflicting substrings
p_3	*FunCall*	{`[\(]`,`[\)]`}	n/a (recursive occurence)
p_4	*Expr*	$\emptyset$	$\emptyset$
p_5	*Expr*	$\emptyset$	$\emptyset$
p_1	*String*	{`[\"][A-Za-z0-9\(\)\,]*[\"]`}	{`[\(]`,`[\)]`}
p_2	*Id*	{`[A-Za-z0-9]+`}	$\emptyset$

Table 13.1: Lexicals inherited by *FunCall*

By no small coincidence, the p_i from the table also happen to constitute the *entire* set of productions we gave as the example grammar. This is of course due to the fact that the example has been deliberately kept simple, leaving out redundant productions. In a more realistic one, there would have likely been more productions. Furthermore, in a C grammar, there would also be conflicts with bracket expressions and typecast operators, since they contain balanced parentheses.)

Given the fact that we have detected clashes based on the chosen candidate signal sequences, we now have two options (see section 11.3 for more information on the related design step):

1. Define catchers for clashing sorts (e.g. *String*) to avoid island/water ambiguities. (This would be relatively easy in the example at hand. The following table shows the associated mapping to islands, water and catchers. This mapping is reflected in the source code of our function call grammar with watery expressions, listed in section A.2.)

p_i	V	type
p_3	*FunCall*	island
p_4	*Expr*	water
p_5	*Expr*	water
p_1	*String*	catcher
p_2	*Id*	water

Table 13.2: Mapping to island/water/catcher

2. Conclude that the candidate signal sequences under consideration are not practical for signaling our COI, and try other signal sequences.

If none of these options are feasible, we may conclude that an island/water/catchers recognition strategy based on signal sequences is not suitable for the construct in question (*FunCall*).

A procedure as described in this chapter may be implemented in a functional programming language (such as ASF). Intermediate results can be cached for efficient performance.

13.5 Discussion

Klusener and Lämmel [KL03] describe a method for semi-automatic derivation of island grammars, which they refer to as *tolerant* grammars, from a given base-line (full) grammar.

The method described in this chapter might have raised the question of where exactly to obtain the productions to be analyzed. Given a full grammar, this is not a problem. However, the reason for using island grammars at all is often that a full grammar does not exist. Then, in the absence of a full grammar, we find ourselves in somewhat of a chicken-and-egg situation.

One approach might be to 'intuitively' construct a prototype set of productions, i.e. a (partial) prototype grammar, which can then be analyzed. In such a case, the method can serve to reinforce the 'creative' process that island grammar design frequently is. This is in fact how the method has been applied in the development of the function call island grammars we have discussed. The language in question was C, with which the author has been familiar for a long time. The prototype grammar from chapter 7 has been created largely from memory, with only casual reference to a formal specification. This was made possible in part because exhaustive detailed knowledge of all possible constructs in the language was unnecessary, due to the tolerant nature of certain water sorts used. Applying the method to the obtained grammar then suggested ways of making this prototype grammar more watery, resulting in the two variants described before, featuring watery expressions and statements.

This does mean however, that before we could become (confidently) aware of the possibilities for increased wateryness by applying lexical inheritance analysis, we have had to resort to a fuller grammar (i.e. the one with the fairly detailed expression grammar) than the one we intended to end up with. So, while the resulting island grammar might be more watery, and thus satisfying in terms of a reduced number of required productions and so on, the exercise was perhaps more academic that it was practical.[2]

In short, while the type of knowledge inferred by the method is certainly useful and in fact required for defining correct island grammars, the method's application may be more of a hassle than it is worth, especially as it may involve specifying a fuller grammar *first*, only to turn it into an island grammar later on, thus defeating the main purpose of the island grammar approach. This problem is also apparent in the method described by Klusener and Lämmel. Still, the formalization seems useful in clarifying the type of analysis that island grammar design necessitates, and perhaps in making some of the relevant questions more

[2]That being said, the process *has* proved instructive, and the watery expressions and statements may serve as future —perhaps intuitive— reference for how to model similar constructs in other languages in a watery fashion.

tangible. If nothing else, it highlights the inherently troublesome nature of their design in more complex cases.

Chapter 14

SGLR performance improvements

14.1 Introduction

Island grammar efficiency Island grammars typically define less productions and sorts than their full counterparts. Therefore, the search space during parsing will likely be smaller, which introduces the possibility of speed gains. However, a sub-optimally constructed island grammar may have negative effects on parsing speed that outweigh these gains. For example, a grammar that allows many ambiguities, or fails to aggregate consecutive occurrences of water-characters into a single sort, will increase the size of the parse trees produced and adversely affect efficiency.

But there are other factors with a possibly deleterious effect on parsing performance. Position information for a parse tree, which tells us on what row and column of a source file a parsed construct occurs, is obviously a rather useful thing to have. However, Verhoeven states in his conclusions that "[...] turning on position information results in unacceptable performance and parse tree sizes". [Ver00] In this chapter, we will be looking at island grammar-specific improvements to the Meta-Environment's parser that might alleviate this complaint by reducing the size of the parse tree at parse-time.

The SGLR parser The ASF+SDF Meta-Environment's parser is called SGLR, for Scannerless Generalized Left-to-right Rightmost derivation parser. SGLR parsing is based on Tomita's GLR parsing algorithm [Tom85], which can parse the entire range of context-free languages. Many other parser generators, such as yacc, generate parsers that can only accept a proper subset of context-free languages.[1]

The nodesucker filter We will discuss a mechanism for reducing the in-memory size of a parse tree, called *node sucking*. Its central idea is illustrated in figure 14.1. It replaces all water-type nodes, which may contain many characters, by very small stub nodes with no lexical content. The water nodes are effectively

[1]GNU Bison, the successor to yacc supports GLR parsing too as of version 1.50.

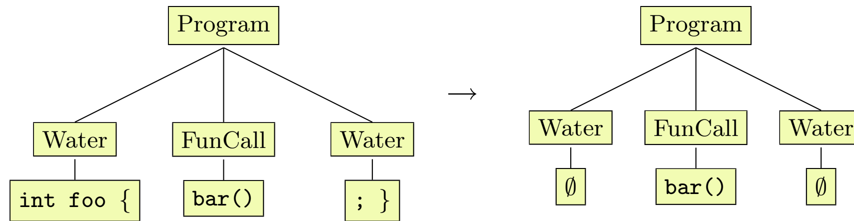


Figure 14.1: Effects of the nodesucker filter

'sucked dry'. We will see how this exploits the space efficiency provided by maximal sharing of ATerms. The advantages of this are:

1. SGLR's memory footprint is reduced.
2. The size of the resulting parse tree is reduced.
3. Because of (1), performance is improved for parses which are so large as to require swapping the data structures that make up the parse tree from resident memory to secondary storage. By reducing the memory footprint by means of parse tree compression, this moment is deferred (or altogether avoided), thus improving performance. (See [Ver00] for example measurements reflecting this problem.)

The filter is not expected to have significant performance benefits other than (3). in fact, since its application involves additional code for checking whether a production is water or not, the parse is expected to incur a (negligible) performance penalty.

14.2 Methods

How GLR parsing works Figure 14.2 shows the dataflow to and from the parser. The first letter of 'SGLR' indicates that it does not have a separate scanner component, unlike most conventional parsers. The lexical syntax is specified in the same SDF specification as the context-free syntax, and the two phases are integrated during parsing. (Actually, upon evaluation of an SDF specification, the lexical and context-free syntax sections are merged into a single set of productions, which are then stored in the parse table. Therefore it would be more accurate to say that there is no separation to begin with.)

GLR parsers exhibit a kind of nondeterminism compared to conventional LR-parsers. Whereas LR-parsers will enter a new state completely determined by the current state and token being processed, GLR parsers may enter one of many states, instead of just a single one. Like LR-parsers, GLR parsers maintain an internal stack. However, because of the 'nondeterminism' just mentioned, the latter employ a more complicated type of data structure than an ordinary stack, called a *Graph Structured Stack* (GSS). To illustrate this, consider the following code example (`funcall.c`):

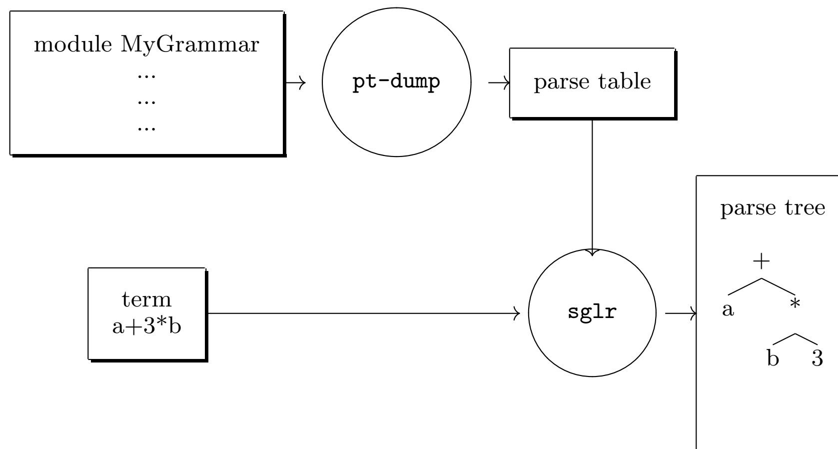


Figure 14.2: dataflow to and from SGLR parser

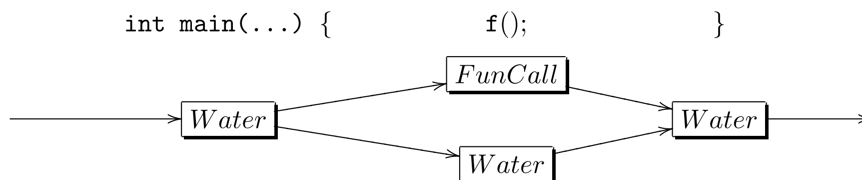


Figure 14.3: GSS forking and joining

```
int main(int argc, char* argv[])
{
  f();
}
```

Let's say we were parsing this over a C-grammar that caused a Water/FunCall ambiguity upon encountering '`f()`'. Now, the parser will fork its stack. Figure 14.3 shows the graph structure of a GSS stack in such a case. Having forked the GSS, the parser may later, upon having scanned more tokens, discover that one of the derivations is not valid, and drop the forked derivation path from the tree. If on the other hand multiple valid simultaneous derivations remain, an ambiguity has been found and will be marked accordingly in the parse forest. (A parse forest is basically a parse tree which may contain ambiguities.) Upon reducing a given state/symbol pair, the parser has the following choices:

1. The parse is forked (may lead to ambiguity later on if alternative is not discarded).

2. Separate paths are further evaluated in parallel without forking or joining.

3. A path turns out to be invalid and is discarded.

4. Separate paths are joined (ambiguity).

Note that the parallel parses share a common prefix and a common suffix. This allows for efficient implementation using ATerms maximal sharing. (This concept is explained below).

14.3 Implementation

Outline One of the design goals of the SGLR parser, as opposed to the SGLR parser *generator*, seems to be to favour simplicity of implementation over speed. Therefore, it provides a good testing ground for a proof of concept implementation which, if successful, might be implemented in the parser generator, which can convert a parser specified in the form of an SDF grammar to C code.

Our proof of concept, outlined in listing 14.1, has been implemented in the function `SG_Reducer()` in `parser.c`[2], right before the fork/join decision described above. Its workings can be described as follows:

- Test if the production considered in the current reduction (`prodl`) is a water production
  - If so, replace the subtree to be inserted by a `stub_tree`; create this subtree if this is the first time the substitution occurs.
  - If not, just create and insert the regular tree `t`.

The `kids` nodes will be automatically destroyed by the ATerm garbage collector if they are no longer referenced from elsewhere, thus freeing memory (see below for more details on ATerms).

On terms and trees The stub tree inserted (see listing 14.1), like all other trees arising during a parse by SGLR, is an ATerm. [BJKO00] It is shown in below in a pretty printed textual form; internally, ATerms are stored in a much more efficient binary form called BAF, for Binary ATerm Format. The `[116]` is an ASCII character code. The other numbers, function arguments to `aprod()`, are numbers representing grammar productions.

```
[
  regular(
    aprod(285),
    [
      regular(
        aprod(285), []
      ),
      regular(
        aprod(287),[116]
      )
    ]
  )
]
```

[2]The full filename is `sdf-bundle-2.4/sglr/libsglr/parser.c`. It can be downloaded as part of the package `sdf-bundle-2.4`, downloadable from `www.meta-environment.org`.

Listing 14.1 Modifications to `parser.c`

```
void SG_Reducer(stack *st0, state s, label prodl,
                ATermList kids, size_t length,
                int attribute)
{

  ...
+  static tree stub_tree = NULL;
+  static ATerm stub_kids = NULL;

  ...

+   /* insert stub subtree in case of water production */
+  if (is_water_prod(prodl)) {
+    /* init: create stub tree */
+    if (stub_tree == NULL) {
+      stub_kids = ATmake("[regular(aprod(285),[regular(↙
    aprod(285),[]),regular(aprod(287),[116])])]");
+      ATprotect(&stub_kids);
+      stub_tree = SG_Apply(table, prodl, (ATermList) ↙
    stub_kids, attribute);
+      /* todo: attribute should also be a stub */
+    }
+    t = stub_tree;
+  }
+  else {
+    /* insert regular subtree */
!    t = SG_Apply(table, prodl, kids, attribute);
+  }

  ...
}
```

A key property of ATerms is that they are implemented to use *maximal sharing*. This means that similar (sub)terms are not duplicated in memory (deep copying), but are duplicated by reference to a single (sub)term (shallow copying). For example, the following ATerm contains the subterm `a(b(c("huey")))` three times:

```
[a(b(c("huey"))), a(b(c("huey"))), "louie", a(b(c("huey")))]
```

In memory, these three occurences are all represented by references to a *single* literal copy of the term, which may potentially save a lot of storage space in real-world situations, since storing a reference to a term is much cheaper than storing the actual term itself. The nodesucker filter exploits this property, as we will later explain.

In the normal situation, i.e. when a non-water production is being reduced, the `kids` are inserted into a subtree `t` by the function call `t = SG_Apply(table, prodl, kids, attribute)`. The subtree `stub_tree` however contains no reference to `kids`. As a result, after execution of the function `SG_Reducer()`, the ATerm `kids` will not be referenced by any tree created during this execution of

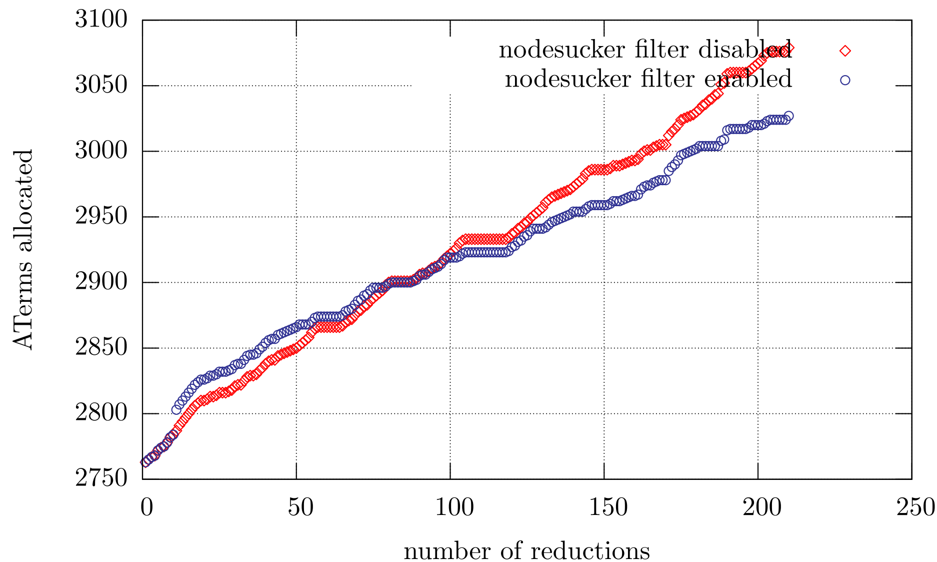


Figure 14.4: ATerms allocated during SGLR parse of `funcall.c` with and without nodesucker filter

and, will consequently be destroyed by the ATerm library's automatic garbage collector if necessary (i.e. if no other references remain).

Measurements Our measurement point occurs in the function `SG_Reducer()`, right before the point where either the regular tree or the stub tree is inserted. Our metric is provided by a call to the function `AT_getAllocatedCount()`, which, as the name suggests, counts the number of ATerms allocated at a given moment. Its output is written to a dataset, in this case a gnuplot file, represented by the file pointer `fp`. This has been implented, in essence, by addition of the following code:

```
AT_alloc_count =  AT_getAllocatedCount();
fprintf(fp, "%d\t%d \n", num_calls,  AT_alloc_count);
```

Here, `num_calls` is a static variable counting the number of calls to the function `SG_Reducer()` and, hence, the number of reductions performed so far.

14.4 Results

Measurements are shown in figure 14.4 and table 14.1.

file	nodesucker	input size	ATerms allocated
funcall.c	off	44	3081
funcall.c	on	44	3029

Table 14.1: Results of nodesucker filter

14.5 Discussion

Graph explanation In the lower left corner of the figure we can see that the two graphs initially overlap. Then, around reduction number 12, the nodesucker-enabled graph initially rises above the regular graph. This local increase in the number of ATerms allocated is caused by the initial creation of the stub tree, which consists of between 20 and 30 ATerms.

After that, we can see that the graphs temporarily have the same shape, although they differ by a constant amount.

Then, the graphs intersect again. Apparently, the ATerm count for the nodesucker version is lower, and this difference continues to increase for every subsequent insertion of a stub tree. This can be explained by the fact that ATerms, of which both the water and the stub trees are made up, are maximally shared [BJKO00] so that multiple insertions of the subtree do not require it to be deep-copied. The difference between the two graphs will widen even further as more stub trees are substituted for water trees. So memory usage is reduced because:

1. The stub tree is very small compared to typtical water trees, which may grow quite large.
2. All water trees are substituted by a reference to only a *single* stub tree, thanks to maximal sharing.

Redundant ATerms will be automatically destroyed by the ATerms library's garbage collector, thus reducing the memory requirements for the parser. All of this results in a space complexity improvement linear in the size of the input.

Conclusion This technique reduces the size of the parse tree during parsing, and therefore of the SGLR's memory usage. This results in a more space-efficient parse in case water sorts are present (and designated as such). In the case of parses that would have to swap to secondary storage without the nodesucker compression, parsing performance may be improved; significantly so, if the parse can now complete entirely in resident memory.

It is especially helpful in deferring the moment at which the in-memory data structures (such as the parse-subtrees arising during parsing and the GSS), have to be swapped to disk. This reduces the reliance on secondary storage, which is an important performance bottleneck in sufficiently long and/or complex parses. [Ver00]

Current limitations

- Position information is not preserved, since the number of characters in the parse tree is different from that in the source text. Possible solutions: run length encoding (RLE) of removed water character sequences. This may be implemented in a number of ways:
    1. Create `kids` ATerm for the subtree's leaves such that the values stored in them represent not ASCII character but the run length of the removed string. (While this would be a fairly elegant solution, it negates (some of) the space and runtime improvements gained by

the method as implemented currently, due to the overhead of creating extra ATerms.

2. Create a location index on non-water parse tree nodes, separate from the parse tree at SGLR runtime. In our example term `funcall.c`, such an index would only have to provide position information for the function call `f()`.

   The concept of using indexes, notably applied in most relational database management systems, is particularly applicable in the case of island grammars, since they typically recognize only a limited number of sorts. Consequently, we would only have to provide position information for relatively few sorts. For the water sorts, whose position information is not relevant since they do not represent constructs of interest, we could suffice with maintaining a counter internally. For every occurence of a non-water sort, we could write the position information to an index, for example a comma separated ASCII file. Such an index could later be used to find extractable constructs quickly, or to reconstruct the original term for unparsing. This could be implemented in a quite straightforward manner, with a constant time complexity overhead for every call to `SG_Reducer()`.

- The function `is_water_prod()` has not been implemented yet; we implement the check by a hardcoded match on the production number, looked up manually from the parse table beforehand. Ideally, this information would be passed, for example, through the SDF specification or on the command line. This would require some additional (straightforward) searching code to determine if a production number `prodl` occurs in a (sorted) list of water production numbers. The time worst-case complexity overhead would be linear in the number of water productions for every call to `SG_Reducer()`.

Alternative optimizations A much more radical and invasive optimization would be to delete water subtrees from the parse tree entirely, during runtime. This would have the advantage of implicit disambiguation, since no water subtree would be able to survive such a filter, which of course means that they could also not be the source of ambiguities. However, we would still have to end up with a well-formed parse tree in some way. While lofty, this goal would require more research and, likely, a lot more implementation effort.

It should be noted that ambiguity filtering is at present non-optimal in general. Quoting from the source code comments in `sglr/libsglr/filters.c`:

> "There are seven separate [post-parse ambiguity] filters:
>
> - cycle detection
> - direct preference
> - indirect preference (heuristic)
> - preference counting (heuristic)
> - injection counting (heuristic)
> - priority

- reject

> The order that these filters are performed in is important because different combinations of filters can remove different trees. The 'strongest' filters should be done first, but this is currently not the case — the priority filter should be done first, but it is actually done last."

Some of these filters require a lot of traversal up and down the tree. If they are in effect, it would more efficient to run the strongest filters first, so as to obtain a minimal parse tree at the soonest possible instance. This would reduce memory overhead, as well as the need for traversal of redundant nodes that will eventually be eliminated from the parse tree anyway.

More details on SGLR parsing and disambiguation can be found in **??**, although some of the mechanisms discussed there have in the meantime been removed from SGLR. Semantic (post-parse) filtering (using ASF) is discussed in [BKMV03].

Finally, it is worth mentioning the existence of *SdfMetz* [AV07], a tool which facilitates the assessment of performance-related aspects of SDF grammars.

Chapter 15

Proposed improvements to the Meta-Environment

15.1 Information in MetaStudio

The documentation of the Meta-Environment is, understandably, not always up to date with the latest changes to the code. While external sources of documentation are available, some of which contain valuable information (e.g. the Stratego/XT manual[1] and various scientific articles), finding where certain features are documented is not always easy. For example, in the course of this research project, I could not get traversal functions to work until I accidentally stumbled upon a relevant paragraph in a thesis about another subject which happened to address the issue explicitly. [Zaa01]

This state of affairs is somewhat unfortunate, since grammar engineering is no exception to the saying that the devil is in the details. More specifically, it is in being *unaware* of certain details, e.g. the programming/specification language in question, the way it is evaluated, the way the IDE operates, and so on.

An example of this in the case of the Meta-Environment would be that an SDF module must always end on an empty line, or it will not be parsed correctly. Another example would be the parse errors caused by not having a suitable Layout sort defined or imported by an ASF specification. In such a case, all that will parse is the obligatory keyword `equations` that starts the specification, and nothing else. While these examples can be sources of confusion when starting out, they will at least cause a visible error, whereas others may be more insidious and lead to errors that will go unnoticed, resulting in incorrect specifications.

Introducing explicit and descriptive warnings to alert the user to shortcomings in their specification can be very helpful in learning how to use the Meta-Environment, and in improving the quality and correctness of ASF+SDF specifications for both inexperienced and advanced users. Given the difficulty of grammar engineering in general, and more particularly that of island grammar engineering, we must consider that certain usability improvements may be crucial to the suitability of the Meta-Environment as a viable solution to island grammar engineering problems. With this in mind, I would like to pose some

[1] http://nix.cs.uu.nl/dist/stratego/strategoxt-manual-unstable-latest/

suggestions for improved information display and state monitoring in MetaStudio:

- Show which tool options are in effect, possibly offering controls to alter them. Of particular interest are SGLR parser flags, such as those that control disambiguation behaviour; we can enable or disable injection counting, priority filtering, reject filtering, etc.
- In case of deprecated features, providing visual clues in the GUI might be prudent. One deprecated feature, prefer/avoid counting, was turned off in the 2.0 versions of the Meta-Environment. While this was documented in the release notes, it had not made its way into the manual yet at the time I was experimenting with this feature, which gave rise to some confusion on my part.
- Show, for example in the import tree view, which module exports which sorts. Conflicts may arise in creating ASF equations if a very general lexical sort has been defined somewhere down the import tree. Hiding unneeded sorts may also be beneficial to the efficiency of the parser, since reducing the number of possible sorts to test for reduces parsing time.
- Provide features for parse tracing; show which rules are matched, which stacks are alive, how much resources are being used, etc. Lex has a debug option (`-d`) option which provides some of this functionality. In particular, it would be useful if there were some way of visualising (e.g. via trees) or tracing erroneous parses up to the point where they failed.
- Provide ASF traversal traces showing e.g. which nodes are visited and which rules are matched.
- Display some basic statistics: LOC, number of productions and sorts per module, etc

Some warnings that would be useful:

- Warn if an SDF specification does not end on an empty line, other than with a general and non-descriptive parse error.
- Warn in case Layout sort is not defined/included. This may especially cause confusion when starting out with ASF.
- Priority warnings (explained below).
- Possibly, warn on trailing spaces and newlines in terms. This is a common source of parse errors is Layout is not defined/included.

Priority warnings Consider the following grammar snippet:

```
context-free syntax
  Expr "+" Expr   -> Expr {left}
  Expr "-" Expr   -> Expr {left}

context-free priorities
  Expr "*" Expr   -> Expr {left} > %% production does not exist
```

```
Expr "+" Expr   -> Expr {left} >
Expr "-" Expr   -> Expr {left} >
Expr "+" Expr   -> Expr {left}   %% duplicate
```

Firstly, SDF will not warn about the fact that the priority for $*$ has no corresponding production. It is very likely that such an occurrence is unintentional and erroneous. Since expression grammars typically are quite large, something like this is easily missed by the grammar engineer.

Secondly, the duplicate specification of the priority for $+$, which cannot exist given the transitive nature of priorities and is consequently erroneous per definition, will generate a parse error for terms such as `a+b+c`. However, SDF should also warn about a specification containing conflicting priorities.

15.2 Suggested new features

We will now describe a number of new features that should prove useful.

Testing framework for SDF Recall from section 1.1 that the correctness of an island grammar depends in particular on the degree to which it admits false positives and false negatives. It is desirable to have some sort of formalism for testing an SDF grammar, particularly for island grammars. I propose the following mechanism to validate an SDF grammar by testing the parsing of certain strings as sorts or their subsorts.

```
strings
  "myfun"                                -> name
  3+x                                    -> expr
  expr                                   -> args
  name "(" args ")"                      -> call
  bla;                                   -> stat

tests
  [Water-1]    stat                      -> Water
  [Expr-1]     expr                      -> Expr

  [FunCall-1]  name "(" args ")"        ~-> Water
  [FunCall-2]  name "(" args ")"         -> FunCall
  [FunCall-3]  name "(" args ")"         -> Expr
  [FunCall-4]  name "(" args ")" ";"     -> Stat
  [FunCall-5]  name "(" args ")" ";"     -> AMBIGUOUS
```

Lowercase words represent string constants. They are a convenient shorthand notation and reduce errors caused by having to retype the same expression. A test $s \rightarrow S$ tests whether the string s be parsed as a term of sort S. The operator `~->` is the logical negation of `->`.

The tests can be resolved by running the strings represented by the concatenation of variables and literals on the left hand sides through the sglr parser with a grammar that defines the sorts on the right hand side, and taking the sort S on the right hand side as the start symbol for each test. If S occurs as a direct or indirect child of a production which also has S on its right hand

side, the test evaluates to *True*. The sort *AMBIGUOUS* is a special case, which evaluates to true if and only if it is a direct or indirect descendant of an ambiguity node in the parse tree.

This can help to systematically test the grammar with respect to certain (suspicious) constructs, especially less elementary ones, for parse errors, ambiguities and false positives and negatives. On the GUI side of things, it would of course be very nice to have a button to add a (sub)term selected in an editing window as a test, possibly commented out initially, to encourage building a set of representative test cases.

A prefer-to disambiguation construct This has been described in section 12.3.1.

The *greedy* attribute SDF has the following notation to implement *greedy* restrictions, also referred to as *follow restrictions* or *prefer longest match*:

```
lexical syntax
  [A-Z][A-Za-z0-9]*          -> Id

lexical restrictions
  Id -/- [A-Za-z0-9]
```

The problem with this notation is that it duplicates information, which makes the specification more error-prone, as it means that edits to the (lexical) production will in most cases require corresponding changes to the follow restriction. In a lot of specifications, the restrictions section will be at a distance of quite a number of lines from the syntax section. Especially since there is no indication in the production's definition that will alert us to the existence of an associated follow restriction, that restriction may be easily overlooked. This in turn would lead to an erroneous specification in case of changes to the sort definition.[2]

Furthermore, the intent of the follow restriction is not immediately obvious from the SDF syntax, which can be especially confusing for novice users. I propose the following alternative notation:

```
lexical syntax
  [A-Z][A-Za-z0-9]*          -> Id {greedy}
```

This would prompt the SDF interpreter to transparently generate code equivalent to that in the first example: a lexical restriction which expresses that the sort *Id* cannot be followed by any character from its repeating tail, in this example: `[A-Za-z0-9]`. This is merely a notational change; the semantics are equivalent to the current notation using the `-/-` operator. This notation however is more concise, more self-explanatory and less error-prone.

[2]This cost me at least an hour when tracing some problem with the minus operator being rejected as part of Expr, which in the end turned out to be the result of the lexical restrictions for Id being incorrect. Ironically, this was after I added this section.

Ambiguity viewer The Meta-Environment GUI currently offers several types of views of a parse tree: without lexicals, with lexicals but without layout, with lexicals and layout or, finally, as a shared tree. The latter view corresponds most closely to how parse trees are stored internally, that is, with maximally shared subterms. Since parse trees can get very large and unwieldy, it may be quite difficult to trace the source of ambiguities visually.

It would be very helpful if there were some way of visualising which nodes in ambiguous subtrees correspond, and which differ. The shared tree view isn't generally much help in this sense. One reason for this is that the lexicals (leaves) at the bottom are no longer sorted in order of occurrence; i.e. the input string is mangled beyond recognition. A suggestion would be to display the trees side by side and highlight the disagreeing nodes, obtaining a visual tree diff. Making tree nodes clickable so they can be folded in or out would also help enormously.

Parser option control It would make sense to allow the user to control certain parser options from inside ASF+SDF specifications, such as which disambiguation filters to use (e.g. priorities, injection counting, prefer/avoid counting). An SDF specification seems like a logical place for such directives since correct behaviour of the grammar may rely on them. (Failing that, it would still be useful to at least expose the default state of the options through the GUI.)

15.3 Other

- Allow graph export to a non-pixel based format (such as GraphViz's dot) from the GUI.
- Make tree nodes clickable and introduce the possibility to hide the subtree rooted at a particular node. (This would also quite convenient for an ambiguity viewer.)
- Visual aterm editor.
- Comments in ASF are somewhat non-obvious. Contrary to SDF itself, which has a built-in comment syntax started by %%, ASF requires comments intended for use inside equations to be defined in its associated SDF specification. The comments defined for the target language specified therein may then be used to comment the ASF equations.
- Implement a scripting language/environment to facilitate integration of external tools and accessing built-in features:
  - SGLR options
  - ASF evaluator options
  - Collecting data for measurements
  - building filter chains from inside the environment
  - filters
  - analyzers (such as SDFMetz),
  - visualizers, etc.
  - Tools such as call graph extractors, RScript, etc.

- While we're at it (i.e. at the scripting idea) an integrated console/interpreter for ASF would be most convenient for testing and experimentation. The file-based terms can be a bit clumsy in certain situations. (This can already be done by using shells such as *bash* (which feature variables) and the Meta-Environment's command line tools to some extent, but we couldn't call that the epitome of usability.) An example session could look something like this (user input is preceded by the prompt `meta>`):

```
meta> $myterm1 = "int main() { foo(); }"

  ok.

meta> extractCalls($myterm1) %% invoke ASF function

  = [funcall("main","foo")]

meta> debug extractCalls($myterm1)

  invoking ASF function 'extractCalls' in debug trace mode:
    ...
    ...
```

  The possibilities are endless and, I would guess for the major part, obvious. We could take hints from some other interpreted environments such as MatLab's interpreter, hugs, QuickBasic, etc, and not derive similar function specifications here.

Chapter 16

Conclusions

16.1 Summary

The general idea behind island grammars is to discard uninteresting constructs by capturing them with, preferably, *lexical* patterns. This is a perceived time-saver, since it eliminates the need to specify syntax in detail, which can indeed be very time consuming. A typical application (and perhaps indeed the most important one) of island grammars is to provide at least *some* grammar in the event that none is available there is none. The practicality of their use which, then, encompasses their development, is a central matter of concern. As we have seen, island grammars may indeed offer the benefits of conciseness ad tolerance. However, these benefits pertain to their *use*, and must be weighed against the complexity of *devising* them.

When developing full grammars, the typical solution to errors in the specification is to refine and/or extend certain productions until they capture the language constructs in question as desired. While generally quite laborious, this process is at least more or less straightforward, and the resulting specification will reflect its purpose explicitly and hence, one hopes, clearly. By contrast, island grammars such as the watery statement grammar of section 10.1.2 are, while quite tolerant, not as self-explanatory as their more explicit counterparts.

While the advantage of not having to specify uninteresting constructs may indeed lead to a significant reduction of specification effort in terms of the number the number of productions, this is a misleading metric, as the potential overall gains also depends on how complicated the analyses are which enable us to arrive at correct island/water definitions sorts. This in turn depends on factors such as the degree to which constructs of interest are nested within uninteresting constructs in the target language, and how well-understood that language is.

For instance, the process used to detect possible literal-clashes between the subsorts of *FunDef*, described in chapter 13, required detailed analysis of parts of the target language. This was used in order to determine how the effects of design decisions concerning (lexical) water sorts, such as which characters they include and exclude, would propagate up the grammar graph. Such questions do not arise in the development of full grammars, since they have no tolerant (water) sorts and, as such, no clashes can occur on account of excessive tolerance.

One may find that the required language analysis, development and testing iterations take more time than simply specifying the constructs in question in detail, i.e. in the style of a full grammar. For certain special cases, e.g. capturing high-level constructs or parsing 'easy' languages, this is not an issue. In the general case however, predicting the potential gains of using an island grammar approach can be difficult.

In short, while we may be able to define terse and correct island grammars with reasonable compositionality properties, the question of their practicality must be carefully considered on a case-by-case basis. Some parts of a language may benefit much from an island grammar approach, while other parts may be too complicated to bother with. The example of an expression grammar from which we can extract function calls, presented in this thesis, is an example in which the merits of an island grammar approach are questionable. In general, assessing the viability of an island grammar approach is difficult and therefore risky.

16.2 Contributions

- Researched, designed and compared island grammars for the recognition of complex constructs of interest.
- Identified and compared some island grammar engineering patterns, and discussed of the issues connected with them.
- Suggested and successfully implemented proof-of-concept efficiency improvements to the SGLR parser.
- Suggested a method for identifying potential clashes between literals in a candidate island grammar, which allows one to reliably define islands, water and catcher productions.

16.3 Future work

- Quantify complexity overhead of designing and testing an island grammar based on certain properties such as the nesting of constructs and target language complexity. This should be probably be considered very difficult.
- Research implicit disambiguation by preventing water nodes from being included in the parse tree.
- Implement improvements suggested in the literature (e.g. [Lee05]), and in this thesis, as part of the Meta-Environment.

Appendix A

Source Code

A.1 Comment extractor source code

Some of the shortcomings of this grammar have been resolved in the function call extractor.

```
module IslandGrammarC

%% functionality:
%% - recognizes function definitions (sort FunDef)
%% - recognizes comments
%%
%% partially implemented:
%% - expressions: only [0-9]+ are valid (sort Expr)
%%

imports Water Comments

exports
  sorts
    Program IslandGlobal WaterGlobal Stat CompoundStat Expr↙
         Id Type FunDecl FunDef FunCall

lexical syntax
  [A-Za-z0-9\_]*                           → Id
  [0-9]+                                   → Expr

lexical restrictions
  Id -/- [a-z0-9\-\_]

context-free syntax
  (IslandGlobal | WaterGlobal)*             → Program

  WATER ";" → WaterGlobal

  IslandComment                            → IslandGlobal {↙
      prefer}
```

```
  FunCall                                 → IslandStat
  Id "(" ")" ";"                          → FunCall

  FunDecl                                 → IslandGlobal {↙
     prefer}
  Id "(" ")" ";"                          → FunDecl {prefer}

  FunDef                                  → IslandGlobal {↙
     prefer}
  Type? Id "(" ")" CompoundStat           → FunDef

  IslandStat                              → Stat {prefer}
  WaterStat                               → Stat {avoid}

  WATER* ";"                              → WaterStat {prefer↙
     }
  WATER* CompoundStat                     → WaterStat {avoid}
  CompoundStat                            → WATER {reject}

  CompoundStat                            → Stat
  "{" (Stat|IslandComment)* "}"           → CompoundStat

  VarDecl → IslandStat {prefer}
  VarDecl → IslandGlobal {prefer}

  Type Id ("[" "]")* ";"                  → VarDecl
  Type Id ("[" "]")* "=" Expr ";"         → VarDecl
  ("int" | "char" | "void") "*"*          → Type

  "char"                                  → WATER {reject}


hiddens
  context-free start-symbols Program



module Comments

imports Layout

exports
  sorts IslandComment DropComment
        Asterisk
        CmtWord CmtBody CmtOpen CmtClose

  lexical syntax
    ~[\ \t\n]+                                  → CmtWord
    "/**" | "/*!"                               → CmtOpen
    "*/"                                        → CmtClose

    "/*" ~[\!\*] ( ~[\*] | Asterisk )* "*/"     → DropComment
    [\*] → Asterisk
```

```
  lexical restrictions
    Asterisk -/- [\/]

    CmtWord -/- ~[\ \t\n]
    CmtOpen -/- ~[\ \t\n]

  context-free restrictions
    LAYOUT? -/- [\/].[\*].~[\!\*]

  context-free syntax
    DropComment → LAYOUT

  CmtOpen CmtBody CmtClose              → IslandComment {↙
     prefer}
  CmtWord*                              → CmtBody
  CmtOpen                               → Drop {reject}
  CmtClose                              → CmtWord {reject}
 module Drop

exports
  sorts Drop

  lexical syntax
    ~[\ \t\n\;\{\}]+ → Drop {avoid}

  lexical restrictions
    Drop -/- ~[\ \t\n\;\{\}]



module Water

exports
  sorts WATER

lexical syntax
  ~[\ \t\n\;\{\}]+                      → WATER {avoid}

lexical restrictions
  WATER -/- ~[\ \t\n\;\{\}]



module Layout

exports

lexical syntax
   [\ \t\n]                             → LAYOUT
context-free restrictions
  LAYOUT? -/- [\ \t\n]
```

```
module ExtractorDoxygen

imports IslandGrammarC

exports
  sorts
    XML Result XMLId RandomXML

context-free start-symbols
  XML Result

lexical syntax
   [A-Za-z0-9]+                               → XMLId

context-free syntax

   "e" "(" Program "," XML* ")"               → Result

   "eToken"
     "(" IslandGlobal | WaterGlobal ")"       → XML
   "eFunCall" "(" CompoundStat ")"
                                              → RandomXML

   "eFunDef" "(" (Stat | IslandComment)*
      "," XML* ")"                            → XML
   "eStat" "(" Stat ")"                       → XML

   "<comment>" CmtBody "</comment>"           → XML
   "<fundecl>" Id "</fundecl>"                → XML

   "<vardecl>" Type Id "</vardecl>"           → XML
   "<water>" WATER* "</water>"                → XML

   "<funbody>" XML* "</funbody>"              → XML

   "<test1 />"                                → XML
   "<test2 />"                                → XML
   "<ignoredstatement />"                     → XML

"<memberdef kind=\"variable\">"
  "<type>"Type"</type>"
  "<name>"Id"</name>"
  ("<initializer>"Expr"</initializer>")?
"</memberdef>"                                → XML

"eFunCall" "(" FunCall ")" → XML
   "<funcall caller" "=" "\"" Id "\">" Id "</funcall>"
                                              → XML

"<memberdef kind=\"function\">"
  "<type>dummytype</type>"
  "<name>"Id"</name>"
  RandomXML
```

```
"</memberdef>"                                  → XML


("<" XMLId (XMLId ("=" "\"" WATER* "\"")?)* ">"
  WATER*
"</" XMLId ">")*
                                                → RandomXML


variables
   "CmtBody"[0-9]*                  → CmtBody
   "CmtOpen"[0-9]*                  → CmtOpen
   "CmtClose"[0-9]*                 → CmtClose

   "IslandComment"[0-9]*            → IslandComment
   "CmtClose"[0-9]*                 → CmtClose

   "IslandGlobalSeq1"[0-9]*         → IslandGlobal*
   "IslandGlobalSeq2"[0-9]*         → {IslandGlobal " "}*

   "IslandGlobal"[0-9]*             → IslandGlobal
   "WATER"[0-9]*                    → WATER
   "CompoundStat"[0-9]*             → CompoundStat

   "TokenSeq"[0-9]*                 → (IslandGlobal | ↙
       WaterGlobal)*
   "Token"[0-9]*                    → (IslandGlobal | ↙
       WaterGlobal)

   "StatOrIslandComment"[0-9]*      → Stat|IslandComment
   "StatOrIslandCommentSeq"[0-9]*   → (Stat|IslandComment)*
   "Stat"[0-9]*                     → Stat

   "XML"[0-9]*                      → XML
   "XMLSeq"[0-9]*                   → XML*

   "Type"[0-9]*                     → Type
   "Id"[0-9]*                       → Id
   "Expr"[0-9]*                     → Expr

   "FunCall"[0-9]*                  → FunCall


equations

[e1]

e(Token TokenSeq, XMLSeq)
= e(TokenSeq,
XMLSeq
eToken(Token)
)

[2] eToken( WATER; ) = <water>WATER</water>
```

```
[3] eToken( CmtOpen CmtBody CmtClose )

=<comment>
  CmtBody
</comment>

[4] eToken( Id(); ) = <fundecl>Id</fundecl>

[5] eToken( Type Id() { StatOrIslandCommentSeq } )
    = eFunDef(StatOrIslandCommentSeq, )


[5c1] eFunDef( Stat StatOrIslandCommentSeq, )
      = eFunDef(StatOrIslandCommentSeq,
        eStat(Stat))

/* discard IslandComment */
[5c2] eFunDef(IslandComment StatOrIslandCommentSeq, XMLSeq)
      = eFunDef(StatOrIslandCommentSeq,
        XMLSeq)

/* parse current statement (default) */
[5c3] eFunDef(Stat StatOrIslandCommentSeq, XMLSeq)
      = eFunDef(StatOrIslandCommentSeq, XMLSeq
    eStat(Stat))

/* terminate the recursion */
[5c4] eFunDef( ,XMLSeq)
=
<funbody>
    XMLSeq
</funbody>

[5d1] eStat(  Id(); ) =<funcall caller="Id">Id</funcall>

[5d2] eStat( Stat ) = <ignoredstatement />

[6] eToken( Type Id; )
=<memberdef kind="variable">
    <type>Type</type>
    <name>Id</name>
</memberdef>


[7] eToken( Type Id = Expr; )

=<memberdef kind="variable">
    <type>Type</type>
    <name>Id</name>
    <initializer>Expr</initializer>
</memberdef>
```

A.2 Function call extractor

```
module IslandGrammarC

%% limitations:
%%
%%   - function pointer calls of the form
%%     '(*func)(1,2)' are not recognized
%%   - Doxygen comments (/**, /*!) are broken
%%   - C++ comments (//_ comments are also not recognized


exports
  imports Layout Water Stat
  sorts
    Program Global IslandGlobal WaterGlobal WaterGlobalWord ↙
        Keyword
  context-free start-symbols Program

lexical syntax
  ~[\ \t\n\;]+                          → WaterGlobalWord {avoid}

lexical restrictions
  WaterGlobalWord -/- ~[\ \t\n\;]

context-free syntax
  Global*                               → Program
  IslandGlobal                          → Global {prefer}
  WaterGlobal                           → Global {avoid}

  IslandComment                         → IslandGlobal {prefer}
  WaterGlobalWord* ";"                  → WaterGlobal {avoid}

  FunDef                                → IslandGlobal {prefer}
  FunDecl                               → IslandGlobal {prefer}
  VarDecl ";"                           → IslandGlobal {prefer}

  CompoundStat                          → Water {reject}

  "inline"? Type?
   Id "(" ParamList ")" CompoundStat    → FunDef

  "else"                                → Water     {reject}
  "case"                                → Water     {reject}
  "goto"                                → Water     {reject}

  "else"                                → Keyword
  "return"                              → Keyword
  "if"                                  → Keyword
  "else"                                → Keyword
  "do"                                  → Keyword
  "while"                               → Keyword
  "switch"                              → Keyword
  "case"                                → Keyword
  "default"                             → Keyword
  "for"                                 → Keyword
  "return"                              → Keyword
  "goto"                                → Keyword

  TypeQualifier                         → Keyword
  StorageClassSpec                      → Keyword
```

```
  Keyword                              → Id       {reject}
  Keyword                              → UserType {reject}



module Stat

imports Comments Decl

hiddens
  sorts CaseLabel GotoLabel

exports

sorts
  Stat WaterStat IslandStat CompoundStat
  XStat XaStat XbStat XaIslandStat XbIslandStat XWaterStat

context-free syntax

  Water*                               → XWaterStat {prefer}

  "{" (Stat|IslandComment)* "}"        → CompoundStat
  CompoundStat                         → Stat
  GotoLabel                            → WaterStat
  Id ":"                               → GotoLabel
  Id ":"                               → Water {reject}
  "default:"                           → Stat


  %% - for the following constructs, add {rejects} in main module
  %% - XaIslandStats end in (compound) statements so they don't
  %%   need to be terminated by ";", whereas XbIslandStats do

  "if" "(" Expr ")" Stat
    ("else" Stat)?                    → XbIslandStat

  "while" "(" Expr ")" Stat           → XbIslandStat
  "switch" "(" Expr ")" Stat          → XbIslandStat
  "for"
    "("XStat";"Expr";"XStat?")"
    Stat    → XbIslandStat
  "case" Expr ":"                     → XbIslandStat

  Expr                                → XaIslandStat
  "return" Expr                       → XaIslandStat
  "do" Stat "while" "(" Expr ")"      → XaIslandStat
  VarDecl                             → XaIslandStat {prefer}

  XWaterStat ";"                      → WaterStat
  XaIslandStat ";"                    → IslandStat
  XbIslandStat                        → IslandStat


  %% the X indicates the sort does not have the semicolon
  %% included at the end
  %% eg: "bla"     → XStat
  %%     "bla" ";" → Stat
  %%
  %% group a: need to be terminated by ";"
  %% group b: end on (compound) statements, don't need a ↙
     terminating ";"
```

```
  XaIslandStat                        → XaStat {prefer}
  XbIslandStat                        → XbStat {prefer}
  XWaterStat                          → XaStat {avoid}
  XaStat ";"                          → Stat
  XbStat                              → Stat
  %% this production provides access to
  %% the semicolon-less version of all statements
  XaStat | XbStat                     → XStat



module Expr

imports Strings FunCall Type

hiddens
  sorts CharConst Digits NatConst Subscript Id
  context-free start-symbols Expr

exports
  sorts Expr Id CastOperator

lexical syntax
  [0-9]+                                             → Digits
  [A-Za-z\_][A-Za-z\_0-9]*                           → Id
  [L]? [\'] (([\\]~[])|~[\\\'])+ [\']                → CharConst


lexical restrictions
  Id       -/- [A-Za-z\_0-9]

context-free syntax

  %% atomic expressions
  CharConst                                          → Expr
  ("+"|"-")?"0x"?Digits                              → Expr
  ("+"|"-")?Digits?"."Digits("e"("+"|"-")Digits)?    → Expr
  Id                                                 → Expr
  String                                             → Expr
  FunCall                                            → Expr

  %% compositional expressions
  "(" Expr ")"           → Expr   {bracket}

  Expr "+" Expr          → Expr {left}
  Expr "-" Expr          → Expr {left}
  Expr "*" Expr          → Expr {left}
  Expr "/" Expr          → Expr {left}
  Expr "%" Expr          → Expr {left}
  Expr "^" Expr          → Expr {left}
  Expr "++"              → Expr
  Expr "--"              → Expr
  "++" Expr              → Expr
  "--" Expr              → Expr

  Expr "=" Expr          → Expr {left}
  Expr "+=" Expr         → Expr {left}
  Expr "-=" Expr         → Expr {left}
  Expr "*=" Expr         → Expr {left}
  Expr "/=" Expr         → Expr {left}
  Expr "%=" Expr         → Expr {left}
```

```
  Expr "^=" Expr          → Expr {left}

  Expr "==" Expr          → Expr {left,prefer}
  Expr "!=" Expr          → Expr {left}
  Expr "<" Expr           → Expr {left}
  Expr "<=" Expr          → Expr {left}
  Expr ">" Expr           → Expr {left}
  Expr ">=" Expr          → Expr {left}

  "!" Expr                → Expr

  Expr "&&" Expr          → Expr {left,prefer}
  Expr "||" Expr          → Expr {left,prefer}

  Expr "&" Expr           → Expr {left,avoid}
  Expr "|" Expr           → Expr {left}
  "~" Expr                → Expr
  Expr "&=" Expr          → Expr {left}
  Expr "|=" Expr          → Expr {left}

  Expr "<<" Expr          → Expr {left,prefer}
  Expr ">>" Expr          → Expr {left,prefer}

  "&" Expr                → Expr
  "*" Expr                → Expr
  Expr "→" Expr           → Expr {left}
  Expr "." Expr           → Expr {left}

  "[" Expr "]"            → Subscript
  Expr Subscript          → Expr

  "(" Type ")"            → CastOperator {avoid}
  CastOperator Expr       → Expr {avoid}

  Expr "?" Expr ":" Expr  → Expr


context-free priorities

  Expr "." Expr           → Expr {left}          >
  Expr "→" Expr           → Expr {left}          >
  Expr Subscript          → Expr >

  Expr "++"               → Expr                 >
  Expr "--"               → Expr                 >
  "++" Expr               → Expr                 >
  "--" Expr               → Expr                 >
  "!" Expr                → Expr                 >
  "~" Expr                → Expr                 >

  Expr "+" Expr           → Expr {left}          >
  Expr "-" Expr           → Expr {left}          >
  Expr "*" Expr           → Expr {left}          >
  Expr "/" Expr           → Expr {left}          >
  Expr "%" Expr           → Expr {left}          >
  Expr "^" Expr           → Expr {left}          >

  Expr "=" Expr           → Expr {left}          >
  Expr "+=" Expr          → Expr {left}          >
  Expr "-=" Expr          → Expr {left}          >
  Expr "*=" Expr          → Expr {left}          >
  Expr "/=" Expr          → Expr {left}          >
```

```
  Expr "%=" Expr          → Expr {left}          >
  Expr "^=" Expr          → Expr {left}          >

  Expr "==" Expr          → Expr {left,prefer}   >
  Expr "!=" Expr          → Expr {left}          >
  Expr "<" Expr           → Expr {left}          >
  Expr "<=" Expr          → Expr {left}          >
  Expr ">" Expr           → Expr {left}          >
  Expr ">=" Expr          → Expr {left}          >
  Expr "&&" Expr          → Expr {left}          >
  Expr "||" Expr          → Expr {left,prefer}   >
  Expr "&" Expr           → Expr {left,avoid}    >
  Expr "|" Expr           → Expr {left}          >
  Expr "&=" Expr          → Expr {left}          >
  Expr "|=" Expr          → Expr {left}          >
  Expr ">>" Expr          → Expr {left,prefer}   >
  Expr "<<" Expr          → Expr {left,prefer}   >
  "&" Expr                → Expr                 >
  "*" Expr                → Expr                 >
  Expr "?" Expr ":" Expr  → Expr


module Strings


exports
  sorts String

  context-free start-symbols String

  lexical syntax
    "\"" StringChar* "\""  → StringCon
    [\\][\"]               → StringChar
    ~[\"\n]                → StringChar
  context-free syntax
    StringCon+             → String

hiddens
  sorts StringCon StringChar

module Decl

imports Expr

exports
  sorts VarDecl Type UserType ParamList Param WATERPARAM ↙
      ReturnType FunDecl FunDef

lexical syntax
  ~[\,]+                            → WATERPARAM {avoid}

lexical restrictions
  WATERPARAM      -/- ~[\,]

context-free syntax
  Type {(Id ("[" "]")*
    ("=" Expr)?) ","}+              → VarDecl


  "inline"? Type?
    Id "(" ParamList ")" ";"        → FunDecl {prefer}
```

```
  Type Id? ("[" "]")*                  → Param
  "..."                                → Param %% variadic ↙
      functions
  Type "(" "*" Id ("[" "]")*  ")"
    "("   {UserType ","}*   ")"        → Param %% fun ptr

  Type ("[" "]")*                      → ReturnType
  {Param ","}*                         → ParamList

module Type

exports
  sorts
    Type UserType
    StorageClassSpec TypeQualifier


lexical syntax
  [A-Za-z\_][A-Za-z\_0-9]*           → UserType

lexical restrictions
  UserType -/- [A-Za-z\_0-9]

context-free syntax
  StorageClassSpec* TypeQualifier* ("struct"|"union")? UserType  ↙
      "*"*
                                       → Type

  "typedef"                            → StorageClassSpec
  "extern"                             → StorageClassSpec
  "static"                             → StorageClassSpec
  "auto"                               → StorageClassSpec
  "register"                           → StorageClassSpec

  "const"                              → TypeQualifier
  "volatile"                           → TypeQualifier

module FunCall

imports Expr
exports
sorts FunCall

context-free syntax
  Id "(" {Expr ","}* ")"               → FunCall



module Comments

imports  Water

hiddens
  sorts
    Asterisk
    CmtWord CmtBody CmtOpen CmtClose

exports
  sorts IslandComment WaterComment
  context-free start-symbols IslandComment WaterComment
```

```
hiddens
  lexical syntax
    ~[\ \t\n]+                              → CmtWord

exports
  lexical syntax
    "/**" | "/*!"                           → CmtOpen
    "*/"                                    → CmtClose
    "/*" ~[\!\*] ( ~[\*] | Asterisk )* "*/" → WaterComment
    [\*]                                    → Asterisk

  lexical restrictions
    Asterisk -/- [\/]
    CmtWord -/- ~[\ \t\n]
    CmtOpen -/- ~[\ \t\n]

  context-free restrictions
    LAYOUT? -/- [\/].[\*].~[\!\*]

  context-free syntax
    WaterComment                    → LAYOUT

    CmtOpen CmtBody CmtClose        → IslandComment {prefer}
    CmtWord*                        → CmtBody
    CmtOpen                         → Water {reject}
    CmtClose                        → CmtWord {reject}


module Water

exports
  sorts Water

  lexical syntax
    ~[\ \t\n\;\{\}\(\)]+ → Water {avoid}

  lexical restrictions
    Water -/- ~[\ \t\n\;\{\}\(\)]



module Layout
   exports
      lexical syntax
         [\ \t\n]          → LAYOUT
  context-free restrictions
    LAYOUT? -/- [\ \t\n]
```

A.3 Extractor using suckpt

Here, we extract subtrees from the source file using `suckpt`. First, we extract the sort *FunDef*. From each fundef,

1. input: parse tree of source file
2. extract subtrees on sort *FunDef* from the tree
3. add entry in sqlite database for each *FunDef* from the tree
4. for each those *FunDef* subtrees, extract sort *FunCall*

5. for each *FunCall*, add entry in sqlite database for each *FunDef* from the tree

```
METAPATH="/opt/asfsdf-meta-2.0.1RC2-linux-i386.bin.sh"
METABINPATH="${METAPATH}/bin"
CLASSPATH="${METAPATH}/share/aterm-java.jar:${METAPATH}/share/↙
    shared-objects.jar:${METAPATH}/share/jjtraveler.jar:."
BASENAME="SuckPT"
module="Autoload"
term="suckpt_test_1.c.tree.txt"
suck="java -classpath ${CLASSPATH} ${BASENAME}"


# this extraction relies on two facts:
#
# 1) the first Id occurring in a FunDef is the name of the ↙
    function defined (true for ANSI C)
# 2) the first Id occurring in a FunCall is the name of the ↙
    function called (true for ANSI C)

rm -f CALLSDB calls.sql
echo "create table calls(caller varchar(100), called varchar(100))↙
    ;">>calls.sql
echo "create table defs(name varchar(100));">>calls.sql

# suck FunDefs
rm FunDef.tree.txt.[0-9]*
${suck} suckpt_test_1.c.tree.txt FunDef

for fundef in FunDef.tree.txt.[0-9]*; do
  # suck Id from FunDefs to obtain name of calling function
  rm Id.tree.txt.[0-9]*
  ${suck} $fundef Id;
  callerId=`${METABINPATH}/unparsePT -i Id.tree.txt.1`

  #echo ============================
  #echo $callerId
  #echo ============================

  callerId=`${METABINPATH}/unparsePT -i Id.tree.txt.1`
  callId="insert into defs values ('`${METABINPATH}/unparsePT -i ↙
      Id.tree.txt.1`');"
  echo $callId >> calls.sql

  # suck FunCalls from  FunDefs
  rm FunCall.tree.txt.[0-9]*
  ${suck} $fundef FunCall;
  for funcall in FunCall.tree.txt.[0-9]*; do
    ${suck} $funcall Id;
    #callId="  --> `${METABINPATH}/unparsePT -i $funcall`"  # name↙
         + arguments
    callId="   ($callerId, `${METABINPATH}/unparsePT -i Id.tree.↙
        txt.1`)"
    echo $callId
    callId="insert into calls values ('$callerId','`${METABINPATH↙
        }/unparsePT -i Id.tree.txt.1`');"
    echo $callId >> calls.sql
  done
done

cat 'calls.sql' | sqlite CALLSDB
```

```
echo 'select caller,called from calls, defs where called=name;' | ↙
    sqlite CALLSDB
```

A.4 SuckPT

```
/**
 * SuckPT.java
 *
 * @author E.J. Post
 *
 * resources:
 *   - http://homepages.cwi.nl/~daybuild/daily-docs/aterm-java/↙
      aterm/package-summary.html
 *   - /path/to/meta/share/x/.../test/TestFib.java:
 *
 */


import java.io.*;
import java.util.*;
import java.lang.Runtime.*;
import aterm.*;


public class SuckPT
{
    int maxdepth = 100;
    int numExtractions = 0;
    int depth = 0;
    ATerm patternApplProdSortToExtract;
    private ATermFactory factory;
    String sortToExtract = "FunCall";
    Boolean verboseFlag = false;
    Boolean dumpUnparsedFlag = false;
    Boolean writeToFileFlag = true;

    public static final void main(String[] args) throws ↙
        IOException
    {
        SuckPT mysucker = new SuckPT(args);
        //System.out.println("all done.");
    }


    public SuckPT(String[] args) throws IOException
    {
        if (args.length < 2) {
            System.out.print("suckpt 0.1a (c) 2007  Erik Post, ↙
                University of Amsterdam <erik@shinsetsu.nl>\n\n");
            System.out.println("usage: suckpt FILENAME SORT\n\↙
                nThis will suck _all_ occurences of SORT from the ↙
                tree in FILENAME");
            return;
        }
        // parse command line arguments
        for (int i=0; i < args.length; i++) {
            String arg = args[i];
            if (arg.equals("-v")) {
                verboseFlag = true;
            }
```

```
        }
        String filename = args[0];
        sortToExtract = args[1];

        if (args.length >= 3) {
            if(args[2] == "-v") verboseFlag = true;
            else verboseFlag = false;
        }

        if(verboseFlag) {
            System.out.print("suckpt 0.1a (c) 2007  Erik Post, ↙
                University of Amsterdam <erik@shinsetsu.nl>\n\n");
            System.out.println("extracting sort '" +sortToExtract ↙
                +"' from file '" + filename +"'");
        }


        factory = new aterm.pure.PureFactory();
        try {
            //ATerm input = factory.readFromFile(System.in);
            FileInputStream f = new FileInputStream (filename);
            //ATerm input = factory.readFromFile(f);
            dump((ATerm)  factory.readFromFile(f));
            if(verboseFlag) {
                System.out.println("extracted " +numExtractions +"↙
                     occurences of sort '" + sortToExtract +"'");
            }

        } catch (ParseError error) {
            System.err.println("Your input was not a valid term!")↙
                ;
        }
    }

    //
    //  traverse parse tree and process nodes
    //

    void dump(ATerm t)
    {
        //  System.out.print("--dump(<ATerm>) at depth "+depth+": ↙
            --:  " +  t.getType());
        if( depth >= maxdepth) {
            System.err.print("maximum tree depth exceeded.\n");
            System.exit(-1);
        }
        switch(t.getType()) {
        case ATerm.APPL:
            depth++;
            ATerm lhs = null;
            //System.err.print("depth: " + depth +"\n");
            List matches1 = t.match(factory.parse("appl(prod(<term↙
                >,<term>,<term>),<term>)"));
            if (matches1 != null) {
                ATerm targetSort = (ATerm) matches1.get(1);
                lhs = (ATerm) matches1.get(3);
                if(targetSort.toString().equals("cf(sort(\"" + ↙
                    sortToExtract + "\"))")) {
                    // we've found a match
                    System.out.print(numExtractions+"  ");
                    if(true || dumpUnparsedFlag) {
```

```
                        dumpTermUnparsed("parsetree(" + t + ",0)")↙
                            ;
                    }

                    numExtractions ++;
                    if(verboseFlag) {
                        System.out.print("writing to: "+ ↙
                            sortToExtract +".tree.txt." +↙
                            numExtractions +"\n");
                    }
                    if (writeToFileFlag) {
                        try{
                            dumpTreeToFile("parsetree(" + t + ",0)↙
                                ", sortToExtract +".tree.txt."+↙
                                numExtractions);
                        }
                        catch (IOException ex) {
                            System.out.println("error writing to ↙
                                file\n");
                        }
                    }
                }
            } else if ((matches1 = t.match(factory.parse("appl(↙
                list(<term>),<term>)"))) != null) {
                lhs = (ATerm) matches1.get(1);
                //      System.err.print("todo: appl(list())\n");
            } else if ((matches1 = t.match(factory.parse("↙
                parsetree(<term>,<term>)"))) != null) {
                System.err.print("todo: parsetree()\n");
                lhs = ((ATermAppl) t).getArgument(0);
            } else {
                // unknown function
                System.err.print("todo: non-appl(prod),...) APPL ↙
                    with arity "+((ATermAppl) t).getArity()+"\n");
            }
            if (lhs != null) {
                dump(lhs);
            }
            depth--;
            break;

        case ATerm.LIST:
            // recurse into all list elements
            for (int i=0; i<((ATermList) t).getLength(); i++) {
                if(dumpUnparsedFlag) {
                    dumpTermUnparsed("parsetree(" + t + ",0)");
                }
                dump(((ATermList) t).elementAt(i));
            }
            break;
        default:
            break;
        }
    }


    // alternative methods provided by ATerm package:
    //    - void writeToTextFile(java.io.OutputStream stream) ↙
        throws java.io.IOException
    //    - void writeToSharedTextFile(java.io.OutputStream stream)↙
         throws java.io.IOException
    //
```

```
    public void dumpTreeToFile(String parseTree, String fileName) ↙
        throws IOException
    {
        FileOutputStream out; // declare a file output object
        PrintStream p; // declare a print stream object

        try {
            out = new FileOutputStream(fileName);
            p = new PrintStream( out );
            p.print (parseTree);
            p.close();
        }
        catch (Exception e) {
            System.err.println ("Error writing to file " + ↙
                fileName);
        }
    }

    public void dumpTermUnparsed(String parseTree)
    {
        // System.out.println(System.getenv("PATH"));
        try {

            String[] cmd = {
                "/bin/sh",
                "-c",
                "echo '"+parseTree+"'| "+"/opt/asfsdf-meta-2.0.1↙
                    RC2-linux-i386.bin.sh/bin" +"/unparsePT"
            };

            String line;

            Process p = Runtime.getRuntime().exec(cmd);
            BufferedReader input = new BufferedReader(new ↙
                InputStreamReader(p.getInputStream()));
            while ((line = input.readLine()) != null) {
                System.out.println(line);
            }
            input.close();
        }
        catch (Exception err) {
            err.printStackTrace();
        }
    }
}
```

Appendix B

ASF+SDF Meta-Environment problems

B.1 Meta-Environment/MetaStudio issues

The Meta-Environment and the MetaStudio IDE have improved enormously since the first time I worked with them in 1998. Installation, both from source and binary, has also become much, much easier compared to previous releases. This may encourage people to experiment with the Meta-Environment. However, there are still a number factors that hinder its learnability, usability and correct operation. To name a few:

- No warning is reported in case a priority rule does not match a context-free syntax rule. If this is the case, the priority rule is useless, and the user may incorrectly assume the source of an ambiguity is not in the priorities section.
- Undo works across multiple buffers in the IDE. Repeated undos may therefore affect other loaded files that we aren't looking at. This is highly uncommon, useless, and leads to much confusion and errors!
- There exists an ATerm-pretty printer (`aterm-pp`) as part of Stratego/XT, but the Meta-Environment does not include this tool. It might come in handy, particularly in reverse engineering te AsFix format as I have done in the production of suckpt.

Miscellaneous bugs There are also some important bugs. Of course, all software has bugs and the Meta versions used throughout this research project were release candidates, not actual releases. However, for the sake completeness, I will list a number of them. Note that some may have been fixed in the meantime:

- Debug reduce hasn't worked across all the 2.x versions of the Meta-Environment I have tried, up to and including the 2.0.1 release candi-

date. Clicking it results in MetaStudio hanging saying 'Rewriting...' in the status line, and nothing else.

- The tool `dump-productions` gives segmentation faults, e.g. when invoked without command line arguments, or when the parse tables fed to it are too large.
- SGLR (version from 2.0.1RC2) bug: reports an error in the *term* when an incorrect file name for the *parse table* (e.g. 'kukelekuu') is given:

```
#sglr -p kukelekuu  -fe -fi -t -l -o ExtractCalls1.tree -i ↙
    ExtractCalls1.trm
sglr: error in ExtractCalls1.trm: unexpected error
```

- Syntax highlighting is not turned off after an unsuccessful parse, in which case highlighting is both useless (for the most part) and confusing.
- After saving an update to an SDF specification, certain types of changes do notappear in the parse tree; specifically in the case of adding prefer/avoid attributes. (Has been fixed)

Buggy traversal function updates in MetaStudio Scenario: the following rule was updated from within MetaStudio:

```
e(Program,X)         → X {traversal(accu)}
```

to

```
e(Program,Result)    → Result {traversal(accu,bottom-up,continue)}
```

Symptoms: the Issues window in the IDE says: "top-down or bottom up missing in traversal strategy—", followed by a line displaying the previous incarnation of the function that used the sort X (see figure B.1). Another bug, occurring after changing the annotation of the traversal function from bottom-up to top-down:

```
context-free syntax
  e(Program,Result)          → Result {traversal(accu,top-down,↙
      continue)}
equations
  [1] e(MyFunCall, result(MyXList)) = result(MyXList,funcall(↙
      MyFunCall))
  [2] e(MyWater, result(MyXList))   = result(MyXList)

e(bla( doe(,result(funcall()))
```

reduced to:

```
result(funcall(start())
```

The "start" is from a previous edit of the equations. Saving does not seem to update things correctly internally using MetaStudio in RC2.

B.2 Documentation issues

The documentation on the Meta-Environment's website[1] was/is not yet completely up to date with the new version. (Note that the Meta versions used

[1]See *www.meta-environment.org*.

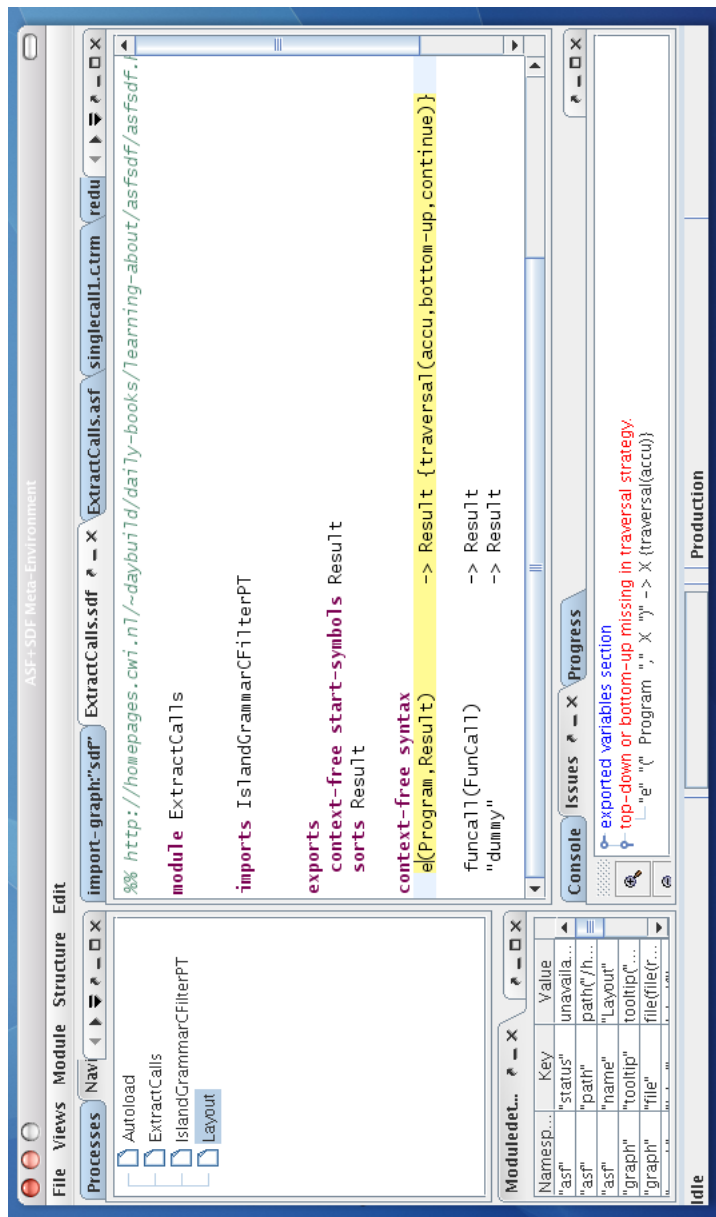


Figure B.1: Meta Enviroment displaying buggy traversal function behaviour

during this project were only release candidates, so all of it is understandably work in progress.)

- MetaStudio has no built-in context-sensitive, hyperlinked help system.
- Some command line tools do not explain what they are for, let alone how they should be used.
- The ToolBus/Java examples and documentation were not yet updated to reflect the new recipe for creating Toolbus tools.
- From the SDF manual: "The function has an attribute named cons with a string as argument. This is used by external tools to give names to constructors in the syntax tree that is built." Statements such as these could do with some examples; the Stratego/XT manual provides some, with drawings of example parse trees and AST's to clarify their use.
- Existing documentation hardly contains any figures showing parse trees and such. (The Stratego/XT online manual on the other hand does contain many enlightening illustrations of example SDF parse trees.)
- A number of non-obvious and inconsistent tool names are used. For instance, with *pt-dump*, does 'pt' mean parse tree or parse table? And what sort of filter does *FilterPT* apply exactly? The answer is: minimization and maximization of sorts, but this is not obvious from the name.
- There is very little mention of other external but potentially helpful tool(set)s such as Stratego/XT or Strafunski in the documentation.
- The Meta-Environment 2.x has avoid/prefer counting turned off by default. The manual still mentioned it as part of the standard disambiguation strategy however. This led to rather a lot of confusion.

Bibliography

[AV07] T.L. Alves, J. Visser. SdfMetz: Extraction of Metrics and Graphs from Syntax Definitions - Tool Demonstration (Draft). (2007)

[BKVV06] M. Bravenboer, K.T. Kalleberg, R. Vermaas, E. Visser. Stratego/XT Tutorial, Examples, and Reference Manual (latest) (draft). Universiteit Utrecht (2006) `http://nix.cs.uu.nl/dist/stratego/strategoxt-manual-unstable-latest/manual/`

[BIK06] O. Bournez, L. Ibaăescu, H. Kirchner. From Chemical Rules to Term Rewriting. LORIA-INRIA, LORIA-UHP, LORIA-CNRS, Campus scientifique BP 239, F-54506 Vandoeuvre-lès-Nancy Cedex, France (2006)

[Lee05] R. van der Leek. Implementation Strategies for Island Grammars. MSc thesis, Technical University of Delft (2005)

[KLV05] P.Klint, R. Lämmel, C. Verhoef. Towards an engineering discipline for grammarware. ACM Transactions on Software Engineering Methodology, Vol. 14, No. 3, ACM Press (2005)

[Kli05] P. Klint. A Tutorial Introduction to RScript — a Relational Approach to Software Analysis (draft) (2005)

[Ver04] R.B. Vermaas. xDoc, an extensible documentation generator. MSc thesis, Utrecht University (2004)

[SCD03] N. Synytskyy, J.R. Cordy, T.R. Dean. Robust Multilingual Parsing Using Island Grammars. Proceedings of the Conference of the Centre For Advanced Studies on Collaborative Research, IBM Press (2003)

[LV03] R. Lämmel, J. Visser. A Strafunski Application Letter. Proceedings of Practical Aspects of Declarative Programming, Springer-Verlag (2003)

[KL03] S. Klusener, R. Lämmel. Deriving tolerant grammars from a baseline grammar. Proceedings of the International Conference on Software Maintenance (2003)

[BKMV03] M.G.J. van den Brand, A.S. Klusener, L. Moonen, J.J. Vinju. Generalized Parsing and Term Rewriting: Semantics Driven Disambiguation. Electronic Notes in Theoretical Computer Science (2003)

[Vis02] J.M.W. Visser. Generic Traversal over Typed Source Code representations. PhD thesis, University of Amsterdam (2002)

[SSV02] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS Legacy Systems. Science of Computer Programming, 45(2-3) (2002)

[Moo02] L. Moonen. Lightweight Impact Analysis using Island Grammars. Workshop on Program Comprehension, IEEE Computer Society Press (June 2002)

[BKV02] M. van den Brand, P. Klint, J.J. Vinju. Term Rewriting with typesafe traversal functions. Centrum voor Wiskunde en Informatica (2002)

[Zaa01] H. Zaadnoordijk. Source code transformations using the new ASF+SDF Meta-Environment. MSc thesis, University of Amsterdam (2001)

[Moo01] L. Moonen. Generating Robust Parsers using Island Grammars. Proceedings of the 8th Working Conference on Reverse Engineering, IEEE Computer Society Press (2001)

[Ver00] E.J. Verhoeven. COBOL island grammars in SDF. MSc thesis, University of Amsterdam (2000)

[BJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, P.A. Olivier. Efficient Annotated Terms. Software, Practice and Experience Vol. 30 No. 3 (2000)

[Deu99] A. van Deursen, T. Kuipers. Building Documentation Generators. Proceedings of the International Conference on Software Maintenance (1999)

[BKV98] M. van den Brand, P. Klint, C. Verhoef. Term Rewriting for Sale. Electronic Notes in Computer Science 15 (1998)

[Vis97b] E. Visser. Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (1997)

[Vis97] E. Visser. Scannerless Generalized LR-parsing (1997)

[Sud97] T.A. Sudkamp. Languages and machines: an introduction to the theory of computer science. 2nd edition, Addison Wesley (1997)

[Kop97] R. Koppler. A Systematic Approach to Fuzzy Parsing. Software: Practice and Experience, Vol. 27, No. 6 (1997)

[BA96] S. Bohner, R. Arnold. Software Change Impact Analysis. IEEE Computer Society Press, 1996.

[Rek92] J.G. Rekers. Parser Generation for Interactive Environments, PhD thesis, University of Amsterdam (1992)

[Chi90] E.J. Chikofsky, J.H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy in IEEE Software. IEEE Computer Society (1990)

[Ben90] K.H. Bennet. An introduction to software maintenance. Information and Software Technology, 12(4) (1990)

[BK89] J.A. Bergstra, J.W. Klop. A Universal Axiom System for Process Specification (1989)

[Aho86] A.V. Aho, R. Sethi, J.D. Ullmann. Compilers: Principles, Techniques and Tools (1986)

[Tom85] M. Tomita. Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems, Kluwer Academic Publishers (1985)