Redhawk: A standalone, language agnostic AST based navigation system

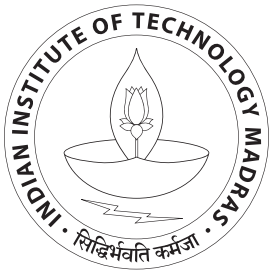
A Project Report

submitted by

PRANESH SRINIVASAN

in partial fulfilment of the requirements
for the award of the degrees of

BACHELOR OF TECHNOLOGY
&
MASTER OF TECHNOLOGY



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

April 2010

THESIS CERTIFICATE

This is to certify that the thesis entitled **Redhawk: A standalone, language agnostic AST based navigation system**, submitted by **Pranesh Srinivasan**, to the Indian Institute of Technology, Madras, for the award of the degrees of **Bachelor of Technology** & **Master of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Shankar Balachandran
Research Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor, Prof. Shankar Balachandran, for allowing me to work on this project. I would also like to thank him for his constant guidance and support without which I would not have been able to finish the project. I think it would also be apt to thank him for taking the course on compilers for our batch which led to my interest in the subject.

Secondly, I would like to thank all my teachers for all that they have taught me. In particular, I would like to thank Prof. R Kalyana Krishnan for being a constant inspiration to me and shaping my programming skills. He has also been constantly available to the student community — we could always walk in and have a chat whenever we wanted to.

Thirdly, I would like to thank my classmate Gautam (BT) for various technical discussions I had with him related to the project and otherwise. I have learnt more from arguments with him (and the ensuing research), than I can recount. I would also like to thank all my friends, who have helped in many ways through the five years. Special thanks are due to Vasanth (Infy), Samhita, NG Srinivas, Aditya Shankar (Gelatin), Arun Chaganty (Slinky), Surya (Virus), Mahitha, Prakash Mohan, Akarsh, Vimal, Siddharth (Lappy), and Kashyap Puranik with whom I have had several technical and philosophical discussions that have shaped me.

Finally, I thank my parents, and my brother, Prashanth, for their constant support and encouragement.

ABSTRACT

Navigating code is the bulk of a programmer's job. Although several good text editors exist, good navigation systems are few and far. Existing standalone tools like ctags and cscope are either low in functionality or support limited languages. On the other hand, most integrated development environments (IDEs) are focused toward one main language, and have only partial support for other languages on an on-demand basis.

This project attempts to build a *standalone*, *scalable* source code navigation system for multiple languages by converting parse trees into a language agnostic abstract syntax tree (L-AST). Two Querying mechanisms are implemented on this L-AST, enabling programmers to query the L-AST for positional information in a effective manner. The *standalone* nature of the tool allows it to be used by editors and IDEs, so that programmers need not leave their environments. Finally, the tool is highly parallelisable, enabling efficient query of large code bases.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS **i**

ABSTRACT **ii**

LIST OF TABLES **vi**

LIST OF FIGURES **vii**

ABBREVIATIONS **viii**

1 Introduction **1**

1.1 Introduction 1

1.1.1 Our approach 2

1.1.2 Use cases and benefits 4

1.2 Organisation of the Thesis 5

2 A Survey of Existing Tools **6**

2.1 Factors considered 6

2.1.1 Independence of a tool from its environment 6

2.1.2 Precision 6

2.1.3 Extensibility 7

2.1.4 Support across multiple languages 7

2.2 Existing Tools 8

2.2.1 Standalone tools 8

2.2.2 Environment Based Tools 9

2.2.3 Key Features of Redhawk 10

2.2.4 Summary 11

3 The L-AST **13**

3.1 Structure of the L-AST . . . 13

3.1.1 Choice of AST over IL as the Language Agnostic Representation . . . 13

3.1.2 Requirements for the L-AST . . . 14

3.1.3 Implementation — A bird's eye view . . . 15

3.2 Support for various Paradigms, Languages, and Constructs . . . 16

3.2.1 A brief tour of programming languages . . . 17

3.2.2 Choice of *C* and *Python* as proof of concept languages . . . 19

3.3 Summary . . . 20

4 Obtaining the L-AST **21**

4.1 Alternatives . . . 21

4.1.1 A custom L-AST parser generator and custom parsers . . . 21

4.1.2 Converting the ASTs obtained by standard parsers . . . 22

4.2 Implementation . . . 23

4.2.1 Parsers Chosen . . . 23

4.2.2 Language Specific Challenges . . . 24

4.2.3 Storing the L-AST . . . 26

5 Operations on the L-AST **31**

5.1 Displaying the L-AST . . . 31

5.1.1 Writer Architecture . . . 31

5.1.2 Writers . . . 32

5.1.3 Aside: Demonstration of the Language Agnostic Nature . . . 35

5.2 Querying . . . 35

5.2.1 The Redhawk XPATH Query Language . . . 37

5.2.2 Implementation of the Redhawk XPATH Query Language 39

5.2.3 Example Redhawk XPATH Queries . . . 41

5.2.4 The Selector API . . . 45

5.2.5 Using the API 48
5.2.6 Parallelisation 50
5.3 Summary 51

6 Conclusions and scope for future work **52**
6.1 Cultural Challenges 52
6.2 Scope for Future Work 53

A Installing and Using Redhawk **54**
A.1 Installing 54
A.1.1 Dependencies 54
A.2 Using Redhawk 55
A.2.1 Getting Help 56
A.2.2 Development (Compile-time) Dependencies 56

B The Vim Plugin **57**

C The XPATH Grammar **58**

D The Node Configuration File and the Node Listing **59**
D.1 The Node Configuration File 59
D.2 Node Listing 61

LIST OF TABLES

2.1 A summary of features across navigation tools. 12

5.1 A path element summary of simpler constructs. 37

5.2 Summary of node queries. 38

5.9 A summary of important modules available via the Redhawk API. 48

5.10 Effect of Parallelisation on `redhawk query '**/DefineFunction'` on the source code of `Django-1.2.1‘` 50

A.1 The `redhawk` commands. 55

D.1 A description of YAML Markup attributes 59

D.2 Listing of all L-AST nodes and their attributes 61

LIST OF FIGURES

1.1 A high level view of Redhawk's approach. 3

3.1 The `Node` hierarchy (only some nodes are shown). 16

3.2 *Python* code to define a `For` L-AST Node. 17

3.3 Redhawk L-AST DSL for defining a `For` L-AST node. 17

4.1 Design of the Language Parsers. 24

4.2 Time taken to perform a module query VS the number of *C* files. 26

4.3 Design of the `ASTFetcher` class. 29

4.4 Procedure to get the L-AST for the source file f (whose key is *key*), with a datastore, d. 30

5.1 Design of the various Writers. 32

5.2 Code Listings of programs to print Hello World 5 times in *C* and *Python* 33

5.3 The `SchemeWriter`'s output of the L-ASTs of `hello_world.c` and `hello_world.py`. The results have been line-aligned to make comparison easier. 34

5.4 The `DotWriter`'s output of the L-ASTs of `hello_world.c` and `hello_world.py`. The figures obtained were rotated in order to fit in the page. . . . 36

5.5 Design of the various Query classes. 40

5.6 Parse a query string q and apply it on a list of L-ASTs, *trees*. 40

5.7 `counter.py` and `stats.c`: Code Listing to demonstrate Redhawk XPATH queries. 42

5.8 Listing of an example *lint* program using the Redhawk API. . . . 49

ABBREVIATIONS

ADT	Abstract Data Type
AST	Abstract Syntax Tree
API	Application Programming Interface
DSL	Domain Specific Language
IDE	Integrated Development Enviornment
L-AST	Language Agnostic Abstract Syntax Tree

CHAPTER 1

Introduction

1.1 Introduction

In today's world of gigantic code bases, and ever changing teams of developers, effective tools for understanding and moving around source code are becoming more and more essential. However, there is no default programmer "toolkit". Every programmer has his/her own way of doing things, and the only common tool is probably `grep`.

Although the programming community is not united in their tools, they are faced with the same set of tasks. These include navigating through a code base, refactoring existing code and writing custom lint/pre-commit scripts to ensure quality of code. Adding to the irony is the language specific issues involved.

C/C++ for example, do not have good source navigation tools. As we shall see in Chapter 2, the two most popular tools for *C* — `ctags`, and `cscope` have their problems. Python, has excellent introspection[1] capabilities, but it is not widely used. Java, on the other hand has excellent support for navigating and refactoring via the many excellent IDEs. However, many programmers do not wish to leave the environments they are already familiar with.

[1] Introspection often involves importing and partly executing the modules involved, which can often lead to side effects, like updating a database, etc..

There is worse ahead. Code bases are becoming larger and more complex. Increasing number of projects are using more than one language to accomplish their goals. Programmers are religious[2] about their environments and editors. There is rising need to ensure quality of code by writing one-off lint scripts. There are more programming languages, and IDEs than ever before.

Also, given N languages and M IDEs, the current method of each IDE having a separate navigation and refactoring system quickly leads to $N \cdot M$ implementations. This leads most IDEs to specialise in one language, and have partial support for other languages on an on-demand basis.

1.1.1 Our approach

The solution we propose for the problem of code navigation is to write a standalone service that clients can consume. This solves the $N \cdot M$ problem and also separates the presentation of navigation information from the logic.

Reusing existing systems is not an option, as we shall see in Chapter 2. For example, the Eclipse IDE's navigation and refactoring system runs in the same thread as its GUI, and therefore, is extremely hard to separate. On the other hand, most existing standalone tools store and present only partial information. We could pry open compilers to extract AST information. However, this is a difficult thing to do. Also, most compilers are written with only correctness and speed in mind. Reuse is generally not a concern.[3]

[2] "Religious" is probably an understatement. One has to only search for "vi emacs holy war" on any search engine to see a plethora of mudslinging.

[3] The `clang` frontend to the `llvm` framework is a notable exception.

This project, titled Redhawk,[4] takes a different approach by using parsers to parse code in various languages, and converting the language specific AST into a language agnostic AST (L-AST).

Such a conversion to a language agnostic AST seems ambitious. This is made possible by keeping in mind that the resulting L-AST is to be used for navigation purposes only[5]. This fact is pivotal. This means, for example, that the semantics of the same L-AST node for different languages can differ slightly[6]. The L-AST captures the syntactic information and the relationships between them for the purposes of navigation. This L-AST can then be queried in a variety of ways for positional information. The crux of our approach is shown in Figure 1.1.

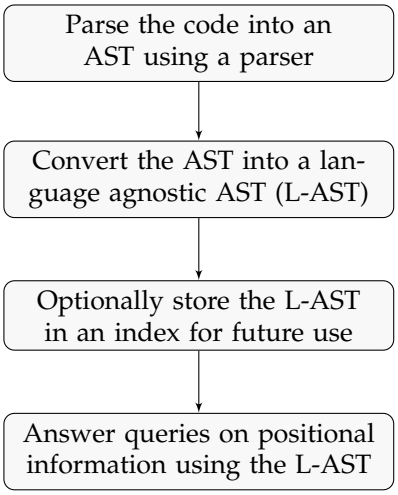


Figure 1.1: A high level view of Redhawk's approach.

The concept of a L-AST for such systems is a novel one. Existing systems like IntelliJ [JetBrains, Retrieved 2011] store language specific ASTs while providing

[4]Hawks are known for their superior visual acuity that aid in navigating to find prey. They were also rated as among the most "intelligent" birds according to avian IQ.

[5]as opposed to say, executing it.

[6]For example, C++ has very different notions of inheritance from *JavaScript* (which uses prototype inheritance). It is fine to represent both kinds of inheritance in the same manner in the L-AST, as the information is to be used only for the purpose of navigation.

the same interface. While useful for the variety of refactoring tasks that IntelliJ allows, this is not as elegant and powerful as having a common data structure like the L-AST. As Alan J Perlis said[Perlis, 1982]:

> "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."

Obtaining the L-AST efficiently and effectively querying it are important requirements of the Redhawk Navigation system. Efficiently obtaining the L-AST is dealt with in Chapter 4. We provide two alternatives for querying the L-AST — one similar to the XPATH query language[Clark *et al.*, 1999], and the other similar to CSS3 Selectors[Çelik *et al.*, 2009]. These are explained in Chapter 5.

1.1.2 Use cases and benefits

There are several use cases for such a tool. We provide a cursory glance here and discuss them in more detail in Chapter 2.

- Simple tasks like jump to declaration/definition.
- Building a system dedicated to code navigation with cross references.
- Querying code with great accuracy (and precision). (Example: Find all classes that inherit from *Foo*, that override the method *Bar*).[7]
- IDEs and editors can use Redhawk to easily enable navigation in a consistent manner across the various languages supported by Redhawk.
- The agnostic nature of the L-AST enables cross language navigation.

There are also a few cultural benefits of this approach:

[7] Chapter 5 and the Appendix are replete with more example queries.

- Consistent tools across languages make it easier to switch languages for developers. This enables them to use the best language for the job at hand.
- Accurate and precise search would help developers in finding exactly what they want to. This would result in a lot of currently difficult tasks becoming easier.[8] Better searches lead to better refactoring, and more automation for common tasks, apart from more demand for better tools.

1.2 Organisation of the Thesis

This thesis is divided into six chapters. Chapter 1 dealt with the motivation of the problem, and an overview of our proposed solution. Chapter 2 looks at existing navigation tools, and describes the key features of Redhawk that set it apart. Chapter 3 deals with the core of the Redhawk Navigation System — the L-AST. Chapter 4 deals with the process of obtaining the L-AST from code. Chapter 5 deals with operations on the L-AST — display and query mechanisms. Finally, Chapter 6 concludes the thesis with a summary and describes the scope for future work.

The chapters mostly deal with the design decisions, trade-offs, and a high level view of the Redhawk Navigation System. The Appendix consists of the more gory details. Appendix A consists of instructions on installing and running Redhawk. Appendix B consists of an editor plugin made using Redhawk. Appendix C consists of the XPATH grammar. Finally, Appendix D lists the L-AST nodes and explains the YAML format used to configure them.

[8] For example, developers would be able to find all assignments involving `NULL` easily. At the same time, this query would be able to filter out incorrect results like `while(p != NULL)`.

CHAPTER 2

A Survey of Existing Tools

In this chapter, we survey existing programming tools. We focus on the following factors for each tool — independence from its environment, precision, extensibility, and support across multiple languages.

2.1 Factors considered

2.1.1 Independence of a tool from its environment

As mentioned in Chapter 1, programmers are religiously tied to their environments. For a tool to be truly useful to all programmers, it has to be standalone. Programmers must be able to access it from their environment of choice in a meaningful manner. Without this, the tool is bound to be tied to a single community.

2.1.2 Precision

For a tool to be effective, the programmer must be able to specify precise tasks to be performed. These tasks could be searching, replacing, or more complex refactoring. High precision ensures that the programmer can perform exactly what he wants to[1], with ease. For example, a high precision search tool would mean that he does

[1] and not much else

not have to wade through a flood of irrelevant results. Similarly, a high precision refactoring tool would not make mistakes when performing a refactoring task, like say, renaming all private methods of a class.

2.1.3 Extensibility

Most software is never finished. Programming tools also tend to undergo a continuous process of refinement depending on the feedback given by the community. Often, a new tool (or workflow) which combines existing tools in the same way is required for repeated application[2], or the edge-case behaviour of a tool needs to be changed[3]. In any case, extensibility is important for a navigation system, as it is impossible for the tool developers to imagine and preempt the needs of the user community.

2.1.4 Support across multiple languages

Programming Languages are chosen not just for their paradigms, but also for the tools accompanying them. The rise of *Java* as a premier programming language has often been attributed to the numerous tools that exist for refactoring, maintaining, and navigation through Java source code, as much as the power of the JVM and the design of the language. Having powerful tools that work uniformly across a multitude of languages gives developers the freedom to truly pick a programming language best suited for the task.

There is also an increasing number of projects being written in multiple lan-

[2] calculating cohesion/coupling by using the results of a search tool, for instance
[3] by filtering or adding results, for instance

guages these days. Most web applications, for instance, require at least three — a server side language, *JavaScript*, and *HTML*[4]. Even performance-oriented programs, like those that find applications in numerical and scientific computing, are written in a lower level language, like *C*, but provide a scripting interface in a higher language like, *Lua*, *Tcl* or *Python*. Language agnosticism of a tool ensures that it can be effectively used across the whole project as a whole.

2.2 Existing Tools

2.2.1 Standalone tools

`ctags`

`Ctags`[Hiebert, Retrieved 2011] is a tag generation tool which aids code navigation, by indexing all language *objects*. Language *objects* include function definitions, identifiers, classes and their parents etc... `ctags` is fast, supports many languages, and is well integrated with most editors.

However, it cannot perform complex queries like finding all callers of a function, or assignments to a particular variable. It also uses a regular expression based heuristic parser (with callbacks). This makes it inherently context insensitive. This leads to problems with ambiguous language grammars. For example, the construct `x::y` in *C++*, can either mean member or method `y` of class `x`, or identifier `y` in namespace `x`. `Ctags` is unable to disambiguate these cases.

[4]It is indeed possible for the server side program to be written in *JavaScript* as well, as is increasingly the case with the advent of *node.js* and the *v8* engine.

In short, `ctags` is fast, and supports multiple languages. However, precise queries cannot be made, and the `ctags` regular expression based parser is context insensitive.

`cscope`

`CScope`[Steffen *et al.*, 1985] is a code browsing tool for the *C* language which uses a fuzzy parser and builds an information database that is able to answer moderately complex queries.

Queries in `cscope` can involve finding a *C* symbol, or global definition, function calls, dependencies[5], or a regular expression pattern. However, its language support is limited to *C*, and is somewhat useful for *C++* and *Java*.

2.2.2 Environment Based Tools

Eclipse[6]

The Eclipse IDE[Eclipse Foundation, Retrieved 2011] has become one of the most used, and respected modern day IDEs, for its excellent *Java* support, and competitive *C/C++* support. The Eclipse IDE has amazing support for navigation, and refactoring. Extremely complex navigation and refactoring is possible through context-based pop-up menus. It accomplishes such complex tasks by maintaining an AST for each program, and providing a common API across the various language specific ASTs.

[5] induced via a #include preprocessor directive

[6] Much of what is said here, can also be said about other the leading IDEs — Netbeans, Visual Studio, etc. . .

However, the navigation and refactoring tools are very difficult to separate from Eclipse. In fact, they run in the main Eclipse thread itself, preventing any process/thread based standalone API from being written.

To summarise, Eclipse has amazing support for *Java*, *C*, *C++*, and partial support for many languages but does so at the cost of being very tightly tied to the environment.

Syntax Directed Editors

Syntax directed editors (or structure editors) like the MENTOR system from INRIA [Donzeau-Gouge *et al.*, 1980] or the ABC Structure Editor for the ABC programming language[Meertens *et al.*, 1992], are aware of the document's underlying structure through some kind of a syntax tree. Syntax directed editors tend to work very well for the languages they understand, and significantly aid programmer experience. The downside is the absence of support for other languages.

2.2.3 Key Features of Redhawk

This is a good point to take a short detour and enumerate the key features of Redhawk.

Firstly, Redhawk runs as a process of its own, and can be used either in a standalone manner or as a service, from within editors. This does not tie Redhawk up to any environment, though integration is possible.

Secondly, it stores the **complete** parse tree[7], enabling very complex queries involving the abstract syntax tree nodes, and relations between them. For example,

[7] in a language agnostic format — L-AST.

it is possible to find the function calls whose fourth argument is `x`, and which are themselves part of an assignment statement. The parse tree is constructed using *robust* full fledged-language parsers for each language, thus preventing incorrect parsing.

Thirdly, it is highly extensible via the *Python* Language. All of Redhawk is available as an API. Third party tools can programmatically use Redhawk for a query, or as part of a much larger tool chain.

Fourthly, implementing a new language in Redhawk is a rather straightforward process. Support for the *C* programming language just took around five hundred lines of code and support for the *Python* programming language only took around seven hundred lines. In return, each language can readily use all the existing scripts and query mechanisms.

However, it must be mentioned that Redhawk does **not** perform static analysis. The resulting L-AST is not meant for execution, and hence certain elements of highly dynamic languages will remain ambiguous.

2.2.4 Summary

A summary of features across navigation tools is shown in Table 2.1.

Tool	Standalone	Complex Query Support	Extensible	Number of languages supported
`grep`	✓	✗	✗	—
`ctags`	✓	✗	✓	41
`cscope`	✓	✓	✗	3
`Eclipse`	✗	✓	✓	3 (partial support for > 7)
`Redhawk`	✓	✓	✓	2

Table 2.1: A summary of features across navigation tools.

CHAPTER 3

The L-AST

In this chapter we focus on the core of Redhawk — the language agnostic representation. We look at two contenders for this language agnostic representation, and reason why the L-AST was chosen over an intermediate language.

We then list the requirements that this language agnostic representation must satisfy in order to be meaningful, followed by how this is achieved by using a domain specific language, and object oriented concepts like inheritance.

Finally, we look at some popular languages to get a picture of some frequently occurring constructs that the L-AST must support. We end with the reasons behind the choice of *C* and *Python* as the proof of concept languages for Redhawk.

3.1 Structure of the L-AST

3.1.1 Choice of AST over IL as the Language Agnostic Representation

Several options exist for the choice of the language agnostic representation. They can be divided into two broad categories — those similar to an *n*-opcode format (also called an intermediate language, or *IL*) like the SSA, and those similar to an abstract syntax tree.

The *n*-opcode format has several advantages — it is terse, efficient, easy to analyse, and several compilers output similar IRs, in various stages. For example, the *GCC* compiler suite outputs a very similar *n*-opcode[Jason Merril, 2003] format across the various supported languages. However, such an IR is far removed from the structure of functional and stack based languages, as well as making the code difficult to visualise. Syntactic (and semantic) information is also often lost during the conversion such IR.

In contrast, a format similar to an AST, is close to the structure of the source code, and enables easy representation of syntax and semantic elements of the source code. It also allows hierarchal querying on the tree thus enabling queries like finding all uses of a variable within a loop. On the downside, compiler frameworks cannot always be directly used, as most of them do not expose the intermediate ASTs, in a simple format.

However, the ability of an AST to maintain the structure of the source code is the most important factor in deciding the internal representation. Such an AST based structure (called the L-AST) forms the core of Redhawk.

3.1.2 Requirements for the L-AST

The nodes in the L-AST must reflect the different constructs and paradigms across various languages. This implies the following important requirements:

1. *A Flexible Structure* — The L-AST inherently lacks a rigid structure. An expression in one language is sometimes a statement in another[1]. Thus, there cannot be a predefined *grammar* for the L-AST.

[1]For example, *printf* in *C* is a function call, whereas *print* is a statement in *Python*.

2. *Extensibility* — Every new language will have some new construct. Therefore, it must be easy to extend the L-AST by adding new nodes as and when required.
3. *Batteries Included* — Common constructs across languages, must already be supported in the L-AST.

3.1.3 Implementation — A bird's eye view

In this subsection, we take a brief look at the implementation of the L-AST to see how the various conditions listed above are realised:

The `Node` hierarchy

The various types of nodes are implemented as classes in *Python*, a highly dynamic language. Every node type inherits from a base `Node` class, as shown in Figure 3.1. A node is therefore, an instance of the corresponding class.

This allows a particular node to have children of any type, thus making the L-AST highly flexible with respect to language grammars, while allowing meaningful relationships among nodes to be enforced when necessary.

Extensibility through a DSL

Although the above design increases flexibility, most of the code involved in implementing a new node is boilerplate cruft, as shown in 3.2.

Writing such code for adding a new node is both time-consuming, and error prone. Instead, we propose a DSL, from which the above *Python* code can be generated. With this DSL, the `For` Node defined above can now be defined succinctly

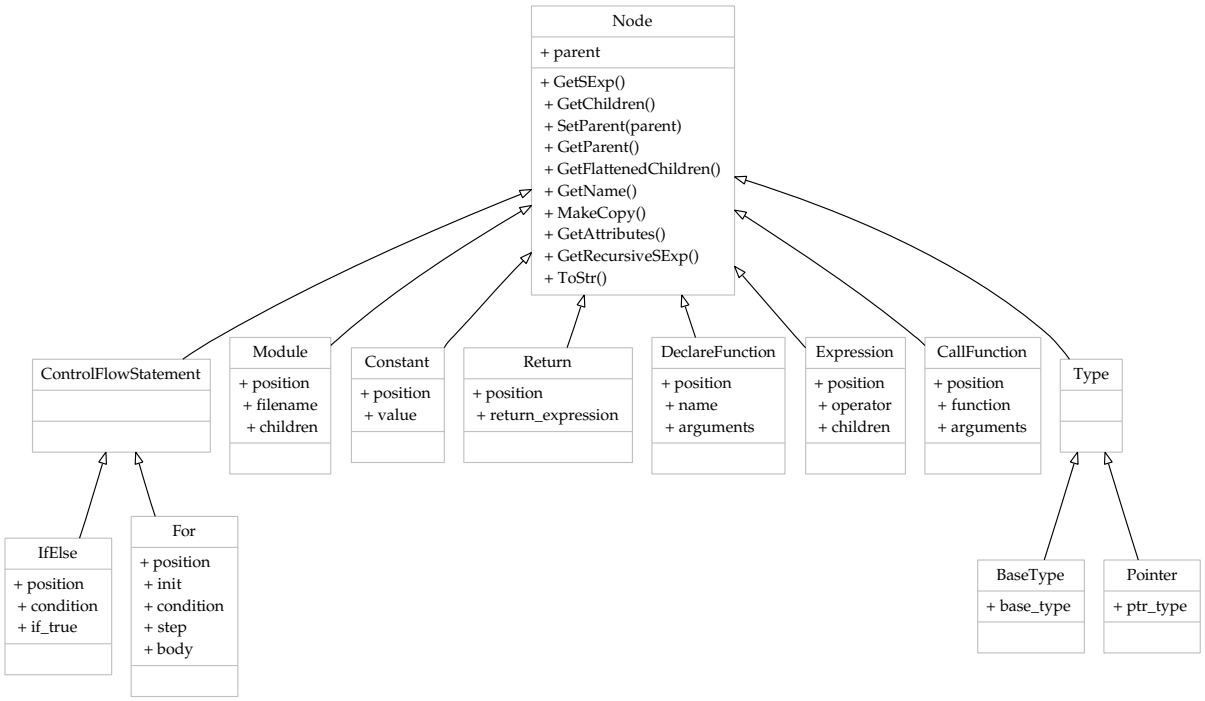


Figure 3.1: The `Node` hierarchy (only some nodes are shown).

as shown in 3.3.

Apart from reducing boilerplate, the above DSL is also more readable. An explanation of this DSL, including the attributes and other features can be found in Appendix D.1.

3.2 Support for various Paradigms, Languages, and Constructs

As discussed in the requirements, the L-AST has to support constructs across different languages with varying paradigms. In this section, we take a brief tour

```
class For(Node):
  """Represents a For Loop construct"""
  def __init__(self, position, init, condition, step, body):
    self.position = position
    self.init = init
    self.condition = condition
    self.step = step
    self.body = body
    return

  def GetChildren(self):
    return  [self.init, self.condition, self.step, self.body]

  def GetSExp(self):
    return ['for', [self.init, self.condition, self.step],
      self.body]
```

Figure 3.2: *Python* code to define a `For` L-AST Node.

```
For:
  docstring: "Represents a For Loop construct"
  args: position, init, condition, step, body
  sexp: ["for", [init, condition, step], body]
  children: init, condition, step, body
```

Figure 3.3: Redhawk L-AST DSL for defining a `For` L-AST node.

of some programming languages, list various constructs[2], and justify the choice of *C* and *Python* proof of concept languages.

3.2.1 A brief tour of programming languages

The Procedural paradigm

C is a classic procedural language. It is fairly close to assembly, supports types, and is compiled. *C* brings along with it, a basket of procedural constructs like

[2] [Yegge, 2010] claims that almost every construct is used in at least two programming languages, and goes on to show some interesting examples.

expressions, iteration (loops), assignment, procedures, and control flow structures like `if-else`, and `switch-case`. Apart from this, *C*, also introduces the notion of ADTs via types, structures and arrays.

The Object Oriented paradigm

Object Oriented languages like *C++*, *Java*, *Smalltalk*, and *Eiffel* offer the abstraction of objects (often in addition to procedural constructs). This includes notions of classes, instances (objects), methods, inheritance, visibility of methods in inherited classes, message passing, contracts, and interfaces.

The Functional paradigm

Functions are first class citizens in functional languages like *Lisp*, *Haskell*, and *ML*. Functional languages heavily differ in most other aspects such as typing, existence of state[3], order of evaluation, support for iteration, and the presence (or absence) of a macro system. Programs written in functional languages tend to make heavy use of recursion and lambda (anonymous functions). As shown in [Abelson *et al.*, 1996], ADTs can be implemented using closures alone. However, most functional languages also provide standard data structures like arrays and lists, and structures or records.

Multiparadigm

Most modern programming languages, such as *Scala* and *Python* are *multiparadigm*. These languages support and encourage the use of two or more paradigms na-

[3] *Haskell* for example does not have a concept of state and assignment.

tively. At times they also provide miscellaneous constructs of their own, such as coroutines, multitasking, continuations, etc. . .

3.2.2 Choice of *C* and *Python* as proof of concept languages

In order to demonstrate the language agnostic nature of the L-AST, we chose *Python* and *C* as proof of concept languages. The reasons for this choice involve the differences between them, as well as the large number of constructs they encompass. We describe them below:

Typing Systems

C is statically typed, i.e., types are assigned at compile time. *Python*, on the other hand, is duck typed[4] i.e., types can only be figured out at run time. Variables in *C* need to be declared before assignment, whereas in *Python* variables are never declared. They come into existence when they are first assigned.

Paradigms

C is a procedural language, whereas *Python* supports procedural, object oriented, and functional abstractions. *Python* has classes, inheritance, lambda functions, generators (streams), and natural support for higher order functions.

Functions are variables in *Python*, and hence are first class citizens. Class and functions can also be defined in any scope (not just at the module level). *Python*

[4] "If it looks like a duck, and quacks like a duck, then it must be a duck." — An object's attributes and methods are inspected rather than its explicit relationship to some type object.

also supports the decorator pattern inherently.

Constructs

C generally offers lower level constructs that *Python*. For example, *C* encourages iteration by indexing, whereas *Python* supports iteration as a concept, via the `in` keyword. *Python* has support for list comprehensions, and provides sets, and dictionaries also as default collections. *C* on the other hand only offers the array as a collection. *Python* also supports coroutines, and exception handling in the language itself, whereas in *C* they have to be implemented using `setjmp`, and `longjmp`.

On the other hand, *C* supports several complex control flow statements like `switch-case` and `goto`, whereas *Python* only supports `if-else`.

In short, *Python* is a very high level, dynamic, interpreted language with functional, object oriented, and procedural abstractions, whereas *C* is a procedural language with low level abstractions. Also between them, most of the constructs described above are covered.

3.3 Summary

In this chapter, we looked at the choice of an AST as the language agnostic representation, conditions any L-AST must satisfy to be truly effective, and how the implementation satisfies these conditions. We also briefly looked at various constructs and paradigms across programming languages, and the reasons for choosing C & Python as prototype languages for the L-AST.

CHAPTER 4

Obtaining the L-AST

In this chapter, we focus on the process of obtaining the L-AST — the core of Redhawk. We look at two alternatives, and justify our choice. We then look at some implementation details, and language specific issues involved. We end the chapter by inspecting the performance of our chosen method and looking at how the L-ASTs are stored to avoid reconstruction.

4.1 Alternatives

We have two methods to obtain the L-AST:

1. Build our own parser generator that creates L-AST parsers: We then write our own parser for each language we wish to support. The common nodes across languages are mapped to the same resulting L-AST node.
2. Use other parsers, and convert the resulting trees: We use existing, robust parsers for each language to obtain a language specific AST, which is then *converted* into a L-AST.

We weigh the advantages and disadvantages of each method below.

4.1.1 A custom L-AST parser generator and custom parsers

Advantages

- *Uniformity*: Every language is treated uniformly — we write a custom parser for that language, using its grammar.

- *Deals directly with grammars*: This method directly deals with the grammar of the language. Changes to the grammar of a language are rare, and well documented. Also, we effectively separate code from data.

Disadvantages

- *Correctness*: Correctness of both the parser generator, as well as the several custom parsers is an issue.
- *Robustness*: Performance and robustness of the custom language parsers are also issues. The custom parsers need to be fast, and have good error handling.
- *Non-standard features*: Many projects use non-standard features of a language, that only a few compilers support. It is difficult to completely replicate all non-standard features in our custom parsers.

4.1.2 Converting the ASTs obtained by standard parsers

Advantages

- *Correctness*: Correctness, and robustness are ensured, with standard parsers.
- *Industrial Strength Parsing*: Standard parsers support language extensions, and are also tuned for performance.
- *Tree Conversion*: Converting the resulting language specific tree into the L-AST is a lot more straightforward that writing a custom parser. It is easy to find missing rules by appropriate error handling. It is also easier to visualise the conversion process than the parsing process[1].

Disadvantages

- *Dependencies*: Dependencies on the chosen parsers for each language is introduced. Also, the language specific AST emitted by these parsers are subject to change far more frequently than the language's grammar.
- *Tedious Conversion*: The conversion of the resulting tree can be a tedious task, that often involves writing quite a bit of boilerplate code.

[1] Most nodes are often directly mapped to corresponding L-AST nodes. The few instances where a node has to be changed and restructured can also be done so in a straightforward manner.

Correctness for the end user is the most important factor however, and outweighs almost everything else. Performance, and Robustness are also fairly important. We hence choose the second method — to use standard parsers and convert the resulting trees.

4.2 Implementation

4.2.1 Parsers Chosen

We use the `pycparser` python package for parsing *C* code. It is fast, stable, and fully C–99 compliant, with upcoming support for GNU C-extensions. It is written in pure *Python*, and uses `ply`, a *Python* re-implementation of lex, and yacc.

For *Python* we use the python parser that comes with the standard *cpython* implementation. The language specific AST is exposed to us through the `ast` module in the python standard library.

The implementation schema is as shown in Figure 4.1. The common `Parser` class in the `redhawk.common.parser` module forms the base class for the `CParser` and the `PythonParser` classes. These inherited classes provide an implementation for the `Parse` method by using `pycparser` and the `ast` modules respectively to parse the code. The `Parse` method returns a *language specific* AST.

The L-AST is obtained by calling the `Convert` method on the language agnostic AST. Each of the inherited classes override the `GetTreeConverterClass` to return the correct converter for the conversion process. The `GetLAST` method in the base class is a useful wrapper around `Parse` followed by a `Convert`.

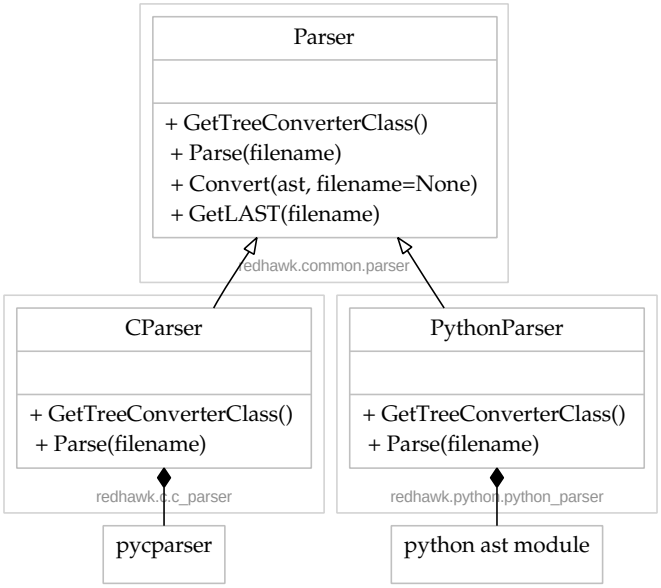


Figure 4.1: Design of the Language Parsers.

4.2.2 Language Specific Challenges

To convert the language specific ASTs to the L-AST in a uniform manner we started with a wish list — of programs and their resulting L-ASTs. Several challenges arose during this conversion. We list some of them below:

C

The `pycparser` module follows a grammar similar to that given in Kerninghan & Ritchie [Kernighan *et al.*, 1988]. Yet, many of the `pycparser` AST nodes are ambiguous. For example, if a `Structure` node in the C-AST does not have a `decls` field, the structure at hand is being referred to. Otherwise, the structure is being defined. This is also true of the `Enum` and `Union` nodes (which represent the `enum` and `union` construct respectively). As another example, the `Case` node only has the first statement in its block. The semantic meaning of a `case` statement however,

refers to the whole block. This creates further issues when we have nested `case` statements.

C also presents the issue of types, and type definitions. A separate hierarchy of types is required (since for example, an array of pointers is not the same as a pointer to arrays). Type definitions of function pointers are also awkward to handle as the various types are dispersed through the language specific AST. These cases have been meticulously handled, but an elegant solution is evasive.

The `pycparser` like most C parsers, parses only preprocessed code. The standard preprocessor `cpp` is used. However, for large projects, complex build systems define custom preprocessor flags. This is hard for Redhawk to replicate, without having to execute the build configuration files.

Python

Python has a much cleaner grammar than *C*. However, it has a few strange features. Firstly, many constructs which would be functions in other languages, are statements in *Python* like `exec`, `eval`, and even `print`. Secondly, comparisons can be chained, like `a <= x < b` which is True if $x \in [a, b)$. These have to be converted into a series of `and` comparisons to be represented in the L-AST.

Thirdly, the syntax for function arguments is rather complex. With presence for normal arguments, keyword arguments, and variable arguments, there is a lot of ambiguity in matching the parameters of a call, to the various types of arguments in the `FunctionArguments` node.

There is also the issue of lambda functions, list comprehensions closures and

scope in *Python*. Many of these constructs create scopes of their own. These need to be handled since a new variable is said to be defined in the current scope in *Python* when it is first assigned in that scope.

4.2.3 Storing the L-AST

Parsing a L-AST from source code is time-consuming. It involves first parsing the code into a language specific AST, and converting it to a L-AST. Repeatedly performing this for every query is not feasible. Instead, we store the L-ASTs in a database[2] after serialising them.

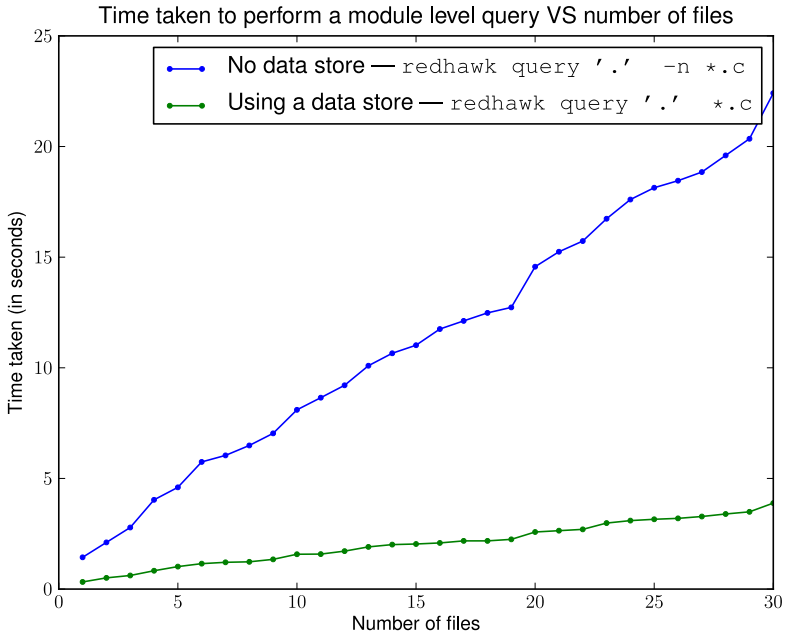


Figure 4.2: Time taken to perform a module query VS the number of *C* files.

The graph in Figure 4.2 shows the performance benefits obtained by using a database to store the L-ASTs. A module query is the simplest query — to find

[2]and cache it for the current run as well.

the top level `Module` Node. This query involves fetching the L-AST, and running through each file, returning the root node[3].

The timing numbers vary from language to language due to the performance difference in the parsers[4]. Hence, we have restricted ourselves to the *C* language[5] to show a meaningful comparison. Each data point was obtained as an average of three runs. These runs were timed with a *hot cache*, i.e., after a couple of untimed runs. The files used were randomly from the source code of the Lua Programming language[Ierusalimschy *et al.*, 1996].

Design of the L-AST store

Currently, the `shelve` module from the *Python* standard library used as the underlying data store, which uses the `pickle` module from the *Python* standard library to serialise the L-ASTs. With even larger code bases like that of say, the linux kernel, we might want to use larger, distributed stores like *Redis* or *Memcached*.

To be able to easily do so, we abstract away the underlying data store as a key value store via the `KeyValueStore` class. This can be seen in Figure 4.3.

Consistency

In order to ensure that we reconstruct the L-AST when the file changes, we also store the SHA–1 [Eastlake and Jones, 2001] hash contents of the file in the store. We match the stored SHA–1 hash against the SHA–1 hash of the file's contents

[3]Finding the root node is an O(1) operation.
[4]Some languages are also inherently easier to parse.
[5]Parsing Python is much faster. A single file takes the order of ~0.1 seconds.

whenever we retrieve a key. If the hashes do not match, we reconstruct the tree.

Since the exact structure of the L-AST may vary from version to version of Redhawk, we also store the version in a special key (`'__redhawk__version__'`). If this key does not match the version number, we delete the database, and on reconstruct, and store the L-ASTs[6].

The module that ties it all up together: `redhawk.common.get_ast`

The `redhawk.common.get_ast` module contains the `ASTFetcher` class and helper functions to get both the language specific ASTs, and the L-AST. The `parser` passed to the `ASTFetcher` during instantiation decides if a language specific AST is returned or the L-AST is returned. The design of the `ASTFetcher` class can be seen in Figure 4.3.

The `ASTFetcher` class is never instantiated directly. Instances of the `ASTFetcher` class are created by calling either `CreateLASTFetcher`, or `CreateLanguageSpecificFetcher`. These factory methods appropriately set the `parser` argument in the creation of the `ASTFetcher`.

The `ASTFetcher` handles the data storage in the data store. Its `GetAST` method gets the AST[7] given the source file, the data store, and the key corresponding to the source file[8]. Simplified pseudo code for the L-AST case is presented in Fig 4.4.

[6] on an on-demand basis.

[7] Either the L-AST or the language specific tree depending on how the `ASTFetcher` class was instantiated.

[8] The key is by default the path to the source file relative to the datastore.

Figure 4.3: Design of the `ASTFetcher` class.

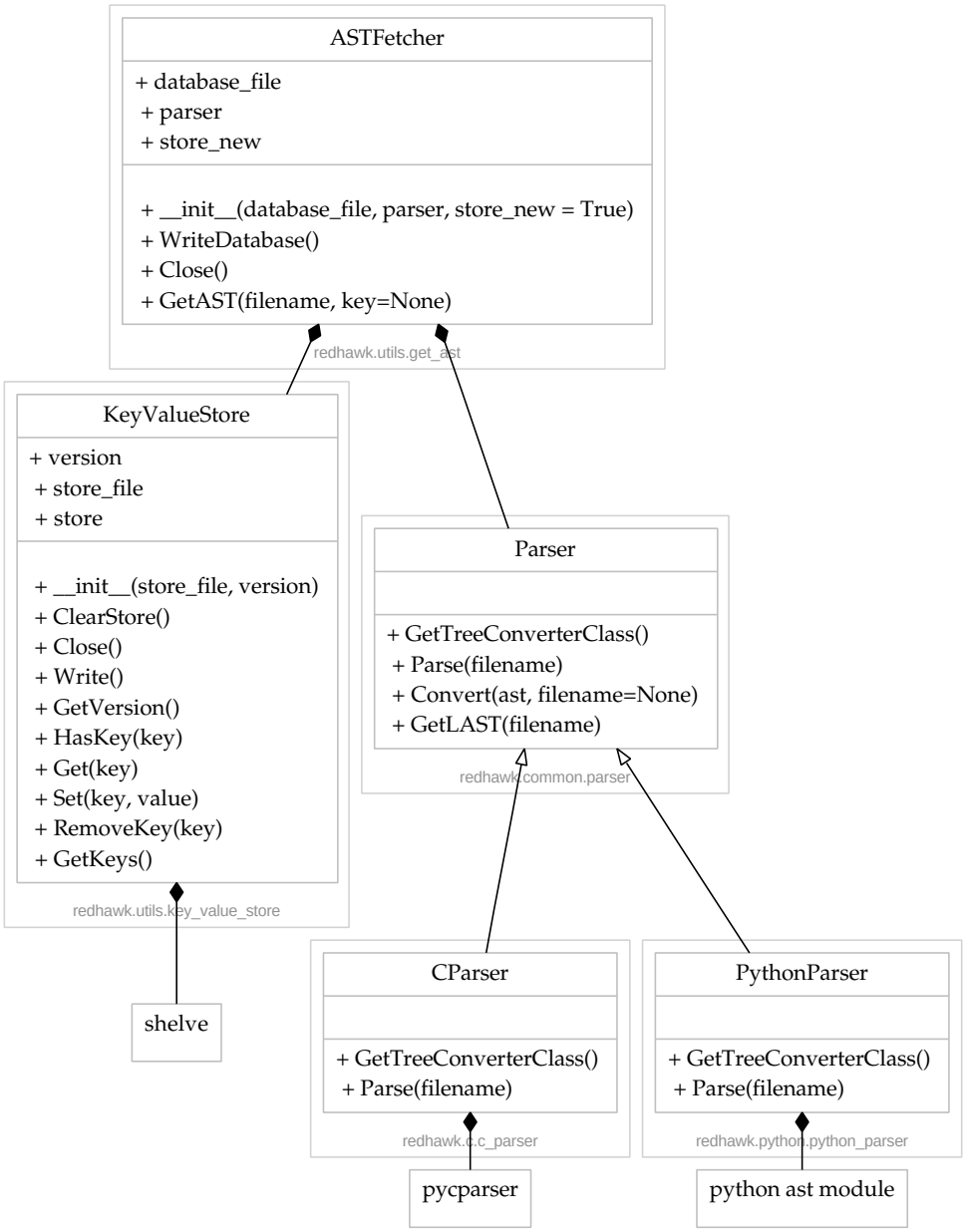


Figure 4.4: Procedure to get the L-AST for the source file f (whose key is *key*), with a datastore, d.

GetLAST(f, d, key)

```
 1  ▷ If datastore is None, return the L-AST by parsing and converting.
 2  if d = None
 3     then return ParseAndConvert(f)
 4
 5  ▷ Ensure that the datastores's version matches current version.
 6  if d.Get(VERSION_KEY) ≠ CURRENT_VERSION
 7     then
 8          d.Close()
 9          RemoveExistingDatabase(d)
10          d ← CreateNewDatabase(CURRENT_VERSION)
11
12  ▷ Get the hash digest of the contents of the given source file.
13  digest ← GetHashDigestOfContents(f)
14
15  ▷ If the datastore has the key and digest matches, return stored AST.
16  if d.HasKey(key)
17     then
18          (stored_digest, stored_ast) ← d.Get(key)
19          if stored_digest = digest
20            then return stored_ast
21
22
23  ▷ Otherwise, we parse the LAST, store it in the datastore for future use, and return.
24  last ← ParseAndConvert(f)
25  d.Set(key, (digest, last))
26  return last
```

CHAPTER 5

Operations on the L-AST

In this chapter, we focus on the operations supported by the L-AST. We begin by looking at two ways to display the L-AST in a human-readable manner. We also use these formats to showcase the language agnostic nature of the L-AST. We then look at two different ways of querying the L-AST — the first is an XPATH-like syntax, and the second is similar to CSS3 Selectors. Finally we look at the *Python* API that Redhawk exposes, and end with a subsection on the ability to query in parallel.

5.1 Displaying the L-AST

The L-AST must often be displayed for the purpose of manual inspection. We propose two such representations — a lisp/scheme like[1] text representation, and a graphical representation.

5.1.1 Writer Architecture

Writers convert the L-AST into human readable forms. As can be seen below in Figure 5.1, the `Writer` base class implements the `WriteTreeToFile` method using

[1] an s-expression like

the pure virtual `WriteTree` method. Writers override the `WriteTree` method, to convert the tree into the specified format.

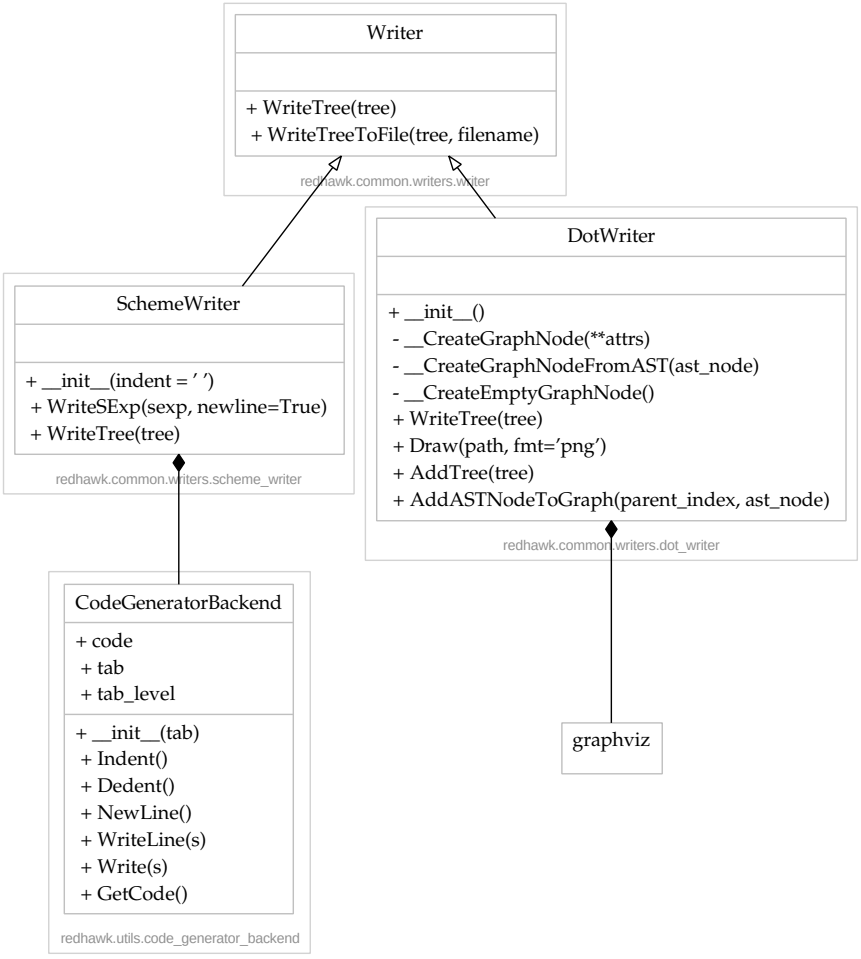


Figure 5.1: Design of the various Writers.

5.1.2 Writers

This is also a good juncture to showcase the true language agnostic nature of the L-AST. We shall consider two very small programs[2], one in *C* (`hello_world.c`), and one in *Python* (`hello_world.py`). They both print `Hello World` 5 times. The

[2]Small only for ease of verifiability.

Code Listing: `hello_world.c`

```
1  void HelloWorld()
2  {
3    printf("Hello World\n");
4  }
5
6  int main()
7  {
8    int i = 0;
9    while(i < 5) {
10     HelloWorld();
11   }
12   return 0;
13 }
```

Code Listing: `hello_world.py`

```
1  def HelloWorld():
2    print "Hello World\n"
3
4  def main():
5    i = 0
6    while i < 5:
7      HelloWorld()
8    return 0
```

Figure 5.2: Code Listings of programs to print Hello World 5 times in *C* and *Python*

programs are shown in Listing 5.2.

We shall see with the help of each writer that the L-AST structure for both of these programs is almost the same[3].

The Scheme Writer

The Scheme Writer converts the L-AST into an s-expression which can be viewed as plain text. The results of running the scheme writer via an invocation of `redhawk show` on `hello_world.c` and `hello_world.py` are shown in Listing 5.3.

The Dot Writer

The Dot Writer converts the L-AST into the `dot` format that `graphviz` can use to generate a pictorial graph. The `pygraphviz` python library is used to interface with `graphviz`.

[3]With the exception of types, due to the difference in their typing systems.

S-Expression representation of the L-AST of `hello_world.c`

```
(define-module helloworld.c
  ((define-function HelloWorld ()
    ((apply printf
      ((constant Hello World
        (:type (base-type string))))))
    (:return_type (base-type void)))

    (define-function main ()
      ((define-variable i
        (:init
          (constant 0
            (:type (base-type int))))
        (:type (base-type int)))

        (while (< i
            (constant 5
              (:type (base-type int))))
          ((apply HelloWorld ())))

        (return
          (constant 0
            (:type (base-type int)))))
      (:return_type (base-type int)))))
```

S-Expression representation of the L-AST of `hello_world.py`

```
(define-module helloworld.py
  ((define-function HelloWorld ()
    ((print
      ((constant Hello World
        (:type (base-type string)))))))


    (define-function main ()
      ((assign i
        (constant 0
          (:type (base-type number))))


        (while (< i
            (constant 5
              (:type (base-type number))))
          ((apply HelloWorld ())))

        (return
          (constant 0
            (:type (base-type number)))))
      )))
```

Figure 5.3: The `SchemeWriter`'s output of the L-ASTs of `hello_world.c` and `hello_world.py`. The results have been line-aligned to make comparison easier.

The results of running the Dot Writer through `redhawk show -i` (or `redhawk show -e`) can be seen in 5.4. Typing information is shown in blue.

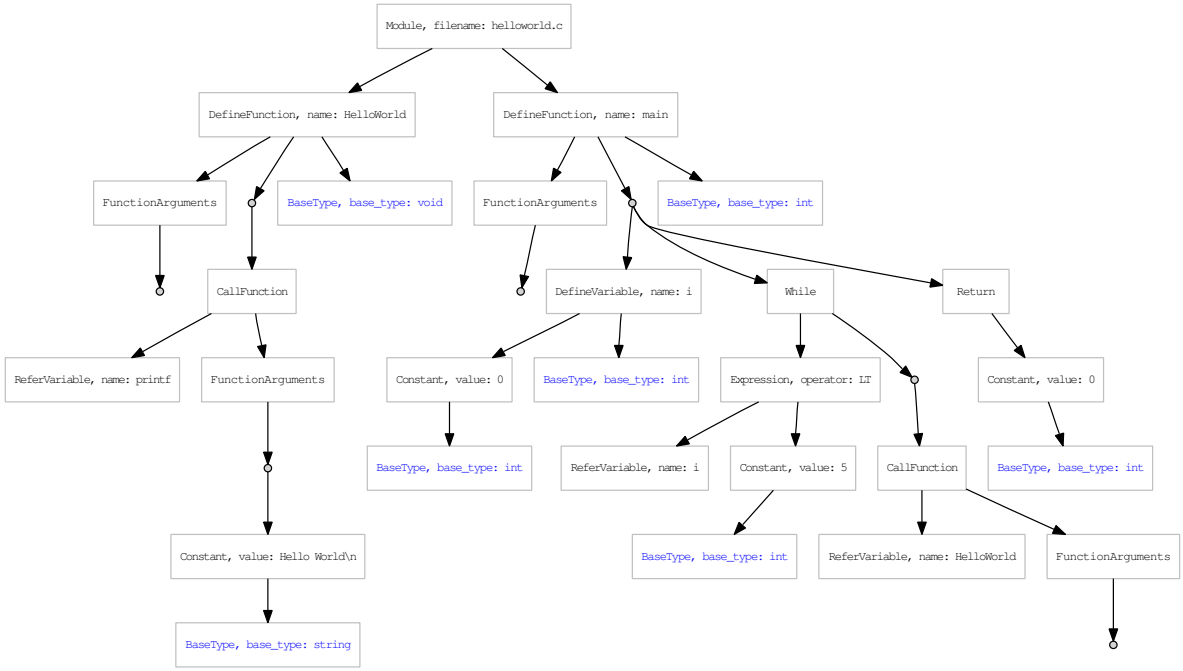
5.1.3 Aside: Demonstration of the Language Agnostic Nature

As can be seen in Figures 5.3 and 5.4, only 3 differences exist between the L-ASTs of `hello_world.c` and `hello_world.py`:

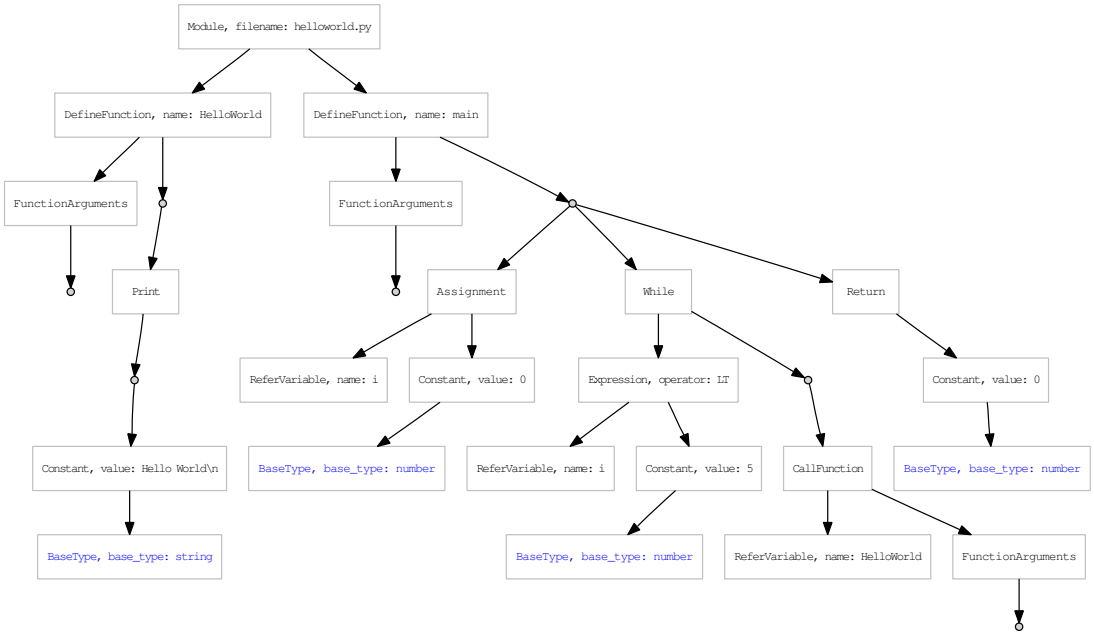
1. `printf` is called in the L-AST of `hello_world.c` whereas the *Print* Statement node is used in the L-AST of `hello_world.py`. This is because `print` is a statement in Python, and we do not want to coerce the structure too much. This can also be seen in Listing 5.2.
2. More typing information in the L-AST of `hello_world.c`. This is due to *C* being a statically typed language. All the extra types shown in the L-AST can also be seen in the original code in Listing 5.2.
3. A `DefineVariable` in the L-AST of the *C* Program compared to an assignment in L-AST of the *Python* program. This is because *Python* does not have the concept of variable *definitions*. In *Python*, a variable is defined, when it first appears on the left hand side of an assignment.

5.2 Querying

Querying the L-AST is the other important part of the Redhawk Navigation System. We provide two methods for querying the L-AST — a query language similar to the XPATH[Clark *et al.*, 1999] query language, and a selector API similar to the CSS3 selector API[Çelik *et al.*, 2009]. Each query mechanism is good for certain kinds of query applications. This section contains the conceptualisation behind each of the query mechanisms.



(a) L-AST of `hello_world.c`



(b) L-AST of `hello_world.py`

Figure 5.4: The `DotWriter`'s output of the L-ASTs of `hello_world.c` and `hello_world.py`. The figures obtained were rotated in order to fit in the page.

5.2.1 The Redhawk XPATH Query Language

We shall loosely call this language the Redhawk XPATH query language, though there it significantly differs from the XPATH language standard. In particular, we do not support axes, namespaces, or operators. On the other hand, we allow arbitrary python expressions in the queries. Our modifications to XPATH are similar to those of the LXML Element Tree XPATH API[Bendersky, 2007].

Brief Introduction to the Redhawk XPATH query language

In its simplest form, a Redhawk XPATH query is one or more path elements separated by slashes (/). A summary of path elements are shown in Table 5.1.

Syntax	Meaning
`Foo`	A *special* case of a Node Query which selects all *child* elements of the node of type `Foo`. For example, `Foo/Bar` selects all nodes of type `Bar` below any node of type `Foo` which is in turn below the root (`Module`).
*	Selects all *child* elements. For example, * selects all children of the root node. Similarly, `*/Foo` would select all nodes of type `Foo` which are grandhildren of the root node.
**	Selects all *descendant* elements. For example, `**/Foo` finds all nodes of type `Foo` *anywhere* in the L-AST.
.	Selects the *current* element. For example, . would match the root node. This is mostly useful only to indicate that the path is a relative one.
..	Selects all *parent* elements. For example `Foo/Bar/..` would select all nodes of type `Foo` which have nodes of type `Bar` as a child.

Table 5.1: A path element summary of simpler constructs.

As shown in the table, `Foo` is the simplest Node query. It selects all *child* elements of the current node of type `Foo`. Node Queries however can get far more

complex. A Node Query can have four parts — we summarize them below in Table 5.2.

Select child nodes based on	Syntax
Type (say `Foo`)	`Foo`
Attribute Value	`@[attribute="value"]`
An arbitrary predicate — say, if its `name` is of even length. (`n` is the default variable. Also, any valid *Python* expression can be used).	`@{len(n.name)%2 == 0}`
Position in parent's children (say 2^{nd} child – index of 1)	`[1]`

Table 5.2: Summary of node queries.

As an example, let us consider the following query:

```
'**/DefineFunction/FunctionArguments/DefineVariable@[name="filename"][2]'
```

This would find nodes of type `DefineVariable` whose `name` field matched the string `"filename"`. Further, it would have to be the third[4] child of its parent, a node of type `FunctionArguments` which would in turn be a child of a node of type `DefineFunction`. This `DefineFunction` could be anywhere in the code.

The `'DefineVariable@[name="filename"][2]'` could have also been written as `'DefineVariable@n.name=="filename"[2]'`[5], using a code block, as opposed to an attribute value matcher.

A grammar of the Redhawk XPATH Query language can be found in Appendix C.

[4]Child positions are zero-indexed.

[5]Note the == in this case. The codeblock must be a valid *Python* expression with the current node as `n`.

5.2.2 Implementation of the Redhawk XPATH Query Language

The Redhawk XPATH queries are parsed and directly run on the L-AST (and not an XML-converted tree). We discuss the parsing and execution below:

Parsing the Query

The Redhawk XPATH queries are parsed using a custom parser in `redhawk.common.xpath`. The custom parser itself is written using a parser combinator[Hutton and Meijer, 1996] library rolled out for this purpose[6] (`redhawk.utils.parser_combinator`). The parsed query can be inspected through an invocation of `redhawk query --show-parsed-query`. For example the query

```
'**/Foo@[bar="quux"]@{n.name is not None}/.././(Bar[2])/*'
```

is parsed as the following list:

1. `StarStarQuery`
2. `NodeMatchQuery`:
   - `node_type` = `'Foo'`
   - `attributes` = `{'bar': 'quux'}`
   - `codegroup` = `'n.name is not None'`
   - `position_list` = `None`
3. `DotDotQuery`
4. `DotQuery`
5. `ChildNodeMatchQuery`: `NodeMatchQuery`:
   - `node_type` = `'Bar'`
   - `attributes` = `{}`
   - `codegroup` = `None`
   - `position_list` = `[2]`
6. `StarQuery`

[6]This was done mainly to avoid additional dependencies. Also, parser combinator libraries are *extremely small* to implement. This one is less than 80 lines of code.

Executing the Query

Each query type above inherits from a base `Query` class, and implements a `Filter` method as shown in Figure 5.5. Each Query's `Filter` method receives a stream of L-AST nodes and filters nodes that matches the query properties.

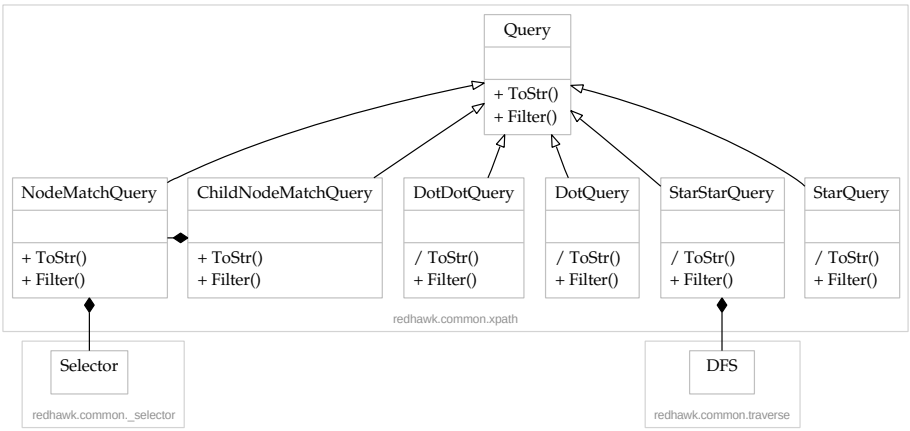


Figure 5.5: Design of the various Query classes.

The overall algorithm for parsing and executing the query in terms of the `Filter` methods is shown in Figure 5.6.

XPath$(trees, q)$

```
1  ▷ Make an iterable out of the list of trees
2  nodes ← iter(trees)
3
4  ▷ Parse the query string, q to get a list of query objects, p
5  p ← ParseXPATH(q)
6
7  ▷ Filter nodes based on the query
8  for i ← 0 to length p
9        do nodes ← p[i].Filter(nodes)
10
11 return nodes
```

Figure 5.6: Parse a query string q and apply it on a list of L-ASTs, *trees*.

5.2.3 Example Redhawk XPATH Queries

In this part, we shall demonstrate some queries on `counter.py` and `stats.c`. The code listing is shown in Figure 5.7. All the queries are run using `redhawk query` on both `counter.py` and `stats.c`.

Example 1 : Functions at *module* level

Query : `'DefineFunction'`

Result :
```
counter.py:19:def CounterIter(init = 0):
counter.py:4:def CounterClosure(init=0):
stats.c:17:float Variance(float *p, int len)
stats.c:5:float Mean(float *p, int len)
stats.c:35:int main()
```

Note that the results are shown in a grep-like format. [7] The functions `Bump` and `Inc` from `counter.py` are correctly missing since we searched for functions only at the module level. To search for functions everywhere, we would have queried with `'**/DefineFunction‘'`.

Example 2 : Closures: Functions Within Functions

Query : `'**/DefineFunction/**/DefineFunction'`

Result :
```
counter.py:6:  def Inc():
```

We observe that `Bump` is not shown, since it is within a class. Queries such as the above illustrate the power of Redhawk, since they cannot be `grep`ped for.

[7] The results, however, are not sorted with respect to line number (though they are guaranteed to be grouped correctly with respect to files). This does not hamper the use of Redhawk for searching and navigation. On the plus side, this makes the querying system a little bit faster. If order is required, a simple invocation of the unix `sort` program should fix this.

counter.py

```
#!/usr/bin/env python
# Three Types of counters:
#   Closures, Classes, Iterators
def CounterClosure(init=0):
  value = [init]
  def Inc():
    value[0] += 1
    return value[0]
  return Inc

class CounterClass:
  def __init__(self, init=0):
    self.value = init

  def Bump(self):
    self.value += 1
    return self.value

def CounterIter(init = 0):
  while True:
    init += 1
    yield init

if __name__ == '__main__':
  c1 = CounterClosure()
  c2 = CounterClass()
  c3 = CounterIter()

  assert(c1() == c2.Bump()
      == c3.next())

  assert(c1() == c2.Bump()
      == c3.next())

  assert(c1() == c2.Bump()
      == c3.next())
```

stats.c

```
/* Mean, and Variance of an array */
#include<stdlib.h>
#include<stdio.h>

float Mean(float *p, int len)
{
  float sum = 0;
  int i;

  for (i = 0; i < len; i++)
    sum += p[i];

  return sum/len;
}


float Variance(float *p, int len)
{
  /* Calculate Variance using
   * E[X^2] - (E[X])^2 */
  float *q = malloc(sizeof(float) * len);
  float variance, mean_p, mean_q;
  int i;

  for(i = 0; i < len; i++)
    q[i] = p[i]*p[i];

  mean_p = Mean(p, len);
  mean_q = Mean(q, len);

  free(q);
  return mean_q - mean_p * mean_p;
}

int main()
{
  float a[] = {1, 2, 3, 4, 5};

  printf("Mean(a) = %f\n", Mean(a, 5));
  printf("Variance(a) = %f\n",
      Variance(a, 5));
  return;
}

```

Figure 5.7: counter.py and stats.c: Code Listing to demonstrate Redhawk XPATH queries.

Example 3 : All assignments affecting the variables `init` or `i`

Query : `'**/Assignment@{n.lvalue.name in ["init", "i"]}'`

Result :
```
counter.py:21:    init += 1
stats.c:10:  for (i = 0; i < len; i++)
stats.c:25:  for(i = 0; i < len; i++)
```

`n.lvalue` is used as we want only the variables on the lvalue (they are *affected*). We could have also written the query as `'**/Assignment/ReferVariable@n.name in ["init", "i"][0]'`. This would have meant finding all `ReferVariable` nodes which were the *first* child (`lvalue`) of the parent `Assignment` node. Further, the `ReferVariable`'s name had to be one of `"init"` or `"i"`.

Example 4 : Arguments named `self`, each of which is the first child of a function definition

Query : `'**/DefineFunction/FunctionArguments/@[name="self"][0]'`

Result :
```
counter.py:15:  def Bump(self):
counter.py:12:  def __init__(self, init=0):
```

This example brings up an important point:

The above query *actually* finds us the `ReferVariables` in the above case, and not the function itself. But this does not hamper our use for navigation and searching when the function definition is in the first line.

Example 5 : All arguments named `self` which are each the last child of a function definition

Query : `'**/DefineFunction/FunctionArguments/@[name="self"][0]'`

Result :

The above query gives us no output!

As can be seen in Appendix D.2 we see that the `FunctionArguments` node has three children — `arguments`, `var_arguments`, `kwd_arguments`, the latter two of which are None everywhere (since no variable or keyword arguments[8] are used). Thus, the children of `FunctionArguments` everywhere in the `counter.py` and `stats.c` files take the form `[arguments, None, None]`. The above query was hence searching for the `kwd_arguments` child of `FunctionArguments` whose `name` was `self`.

What we really want, is the last element of the first element, the arguments list. This can be expressed as follows:

Example 6 : All arguments named `self` which are each the last child of a function definition

Query : `'**/DefineFunction/FunctionArguments/@[name="self"][0, -1]'`

Result : `counter.py:15:  def Bump(self):`

This rightly gives us the `Bump` function.

As has been illustrated the Redhawk XPATH query language is versatile to cover most queries that programmers would need on a day to day basis. For more complex queries, the Selector API exists.

[8] Variable arguments in Python are formal parameters of the form `*bar`, and Keyword arguments are formal parameters of the form `**quux`. A formal paramter of the form `name = value` is a *default* argument and *not* a keyword argument. To sum it up, `def foo(a, b, c=None, **d, *e)` defines a function with two normal arguments `a` and `b`, one default argument `c` with default value `None`, one keyword argument `d`, and one variable argument `e`. [van Rossum and Drake Jr, Retrieved 2011] describes this in more detail.

5.2.4 The Selector API

The Selector API is a more object oriented and verbose version of the CSS3 Selector API[Çelik *et al.*, 2009]. The Selector API is implemented in `redhawk.common.selector`, and uses the bare-bones selector functions defined in `redhawk.common._selector`. The Selector API is used by instantiating selector-objects (or `S-objects`) and applying them on trees.

Instantiating and Applying S-objects

`S-objects` can be used to select some nodes of an AST depending on various constraints. Four common types of constraints involve:

1. Selection based on the type of the node.
2. Selection based on the attributes of the node.
3. Selection based on some custom criteria, i.e, whether a function f, returns True for the given node.
4. A combination of the above.

We show examples for each of the four constraints below:

The following code snippet would define a selector-object, `s` to select nodes of type `Foo`. Running it on a list of trees, `trees`[9], would give us nodes of type `Foo` anywhere in `trees`:

```
s = S(node_type = 'Foo')
```

Similarly, we can create selectors to select based on attributes of the node. `s` in the code snippet below selects node if their `name` attribute has the value `Bar`.

[9] An `s-object`, `s`, can be run on a list of trees, `trees`, by calling it as so — `s(trees)`. This returns an iterator to the list of results.

```
s = S(name = 'Bar')
```

We can also create selectors to select nodes based on any predicate we choose. We do this by passing a function to the `s-object` constructor, `S`. Suppose we wanted to find all function names that did not comply to our coding style guide, which says:

1. All function names should start with a capital letter
2. Function names should not have the '_' character in them.

We could write this function in Python as follows:

```
1  def HasBadName(f):
2    return "_" in f.name or not('A' <= f.name[0] <= 'Z')
```

We may now create a selector which selects all functions that do not conform to our coding style guide as follows:

```
S(node_type = 'DefineFunction', function = HasBadName)
```

Note that we have combined two types of constraints in the above instantiation.

Writing individual functions each time is cumbersome. There is respite, however, since Python provides λ-functions. The above could also now be written as:

```
S(node_type = 'DefineFunction', function =
  lambda n:  "_" in n.name or not('A' <= n.name[0] <= 'Z'))
```

Combining Selectors

Selector-objects objects can also be combined. We provide six means of combination:

1. `s1.And(s2)` returns a *new* `s-object` that selects a node only if *both* `s1` and `s2` select the node.
2. `s1.Or(s2)` returns a *new* new `s-object` that selects a node if *either* `s1` or `s2` select the node.
3. `s1.HasChild(s2)` returns a *new* `s-object` that selects a node, `n`, only if `s1` selects the node, `n`, *and* `s2` selects a child of `n`.
4. `s1.HasDescendant(s2)` returns a *new* `s-object` that selects a node, `n`, only if `s1` selects the node, `n`, *and* `s2` selects some descendant of `n`.
5. `s1.HasParent(s2)` returns a *new* `s-object` that selects a node, `n`, only if `s1` selects the node, `n`, *and* `s2` selects the parent of the node, i.e., (`n.GetParent()`).
6. `s1.HasAncestor(s2)` returns a *new* `s-object` that selects a node, `n`, only if `s1` selects the node, `n`, *and* `s2` selects some ancestor of `n`.

As an example, let us consider the task of finding all private methods in a Python program. Private methods in a Python program are methods whose names start with two underscores — `'__'`. Writing a selector for this condition is easy. We shall call this selector `p`:

```
p = S(node_type = 'DefineFunction',
      function = lambda n:  n.name[:2] == '__')
```

However, for the function chosen above to be a method it must be within a class definition. Since it is the method we want, we can write this using the `HasParent` combinator. The final selector is as follows:

```
p.HasParent(S(node_type = 'DefineClass'))
```

5.2.5 Using the API

All functionality in Redhawk is exposed via the Redhawk API. The Redhawk API can be used by simply importing the required modules in any *Python* program once Redhawk is installed (see Appendix A.1). We list some of the more useful modules in Table 5.9, before delving into an example:

Table 5.9: A summary of important modules available via the Redhawk API.

Module	Functionality
`redhawk.common.get_ast`	Contains functions to fetch the L-AST as described in Chapter 4.
`redhawk.scripts.script_util`	Contains functions to get the currently active database, and convert a filename to a unique key.
`redhawk.common.selector`	The Selector API for selecting nodes.
`redhawk.common.xpath`	The Redhawk XPATH API for querying the L-AST.
`redhawk.common.format_position`	Contains useful functions to print the context of a node in it's file.

We illustrate a use of the Redhawk API by finishing the example started in the last section — to find function names which do not conform to our coding standard. The entire program is shown in Listing 5.8.

Lines 2, 3 and 4 import the relevant modules of the redhawk API. Only the `S` class from the selector module is imported. Line 6 deals with importing the `sys` standard module, required for `sys.argv`, the command line parameters passed in. Lines 8—12 set the filename as the first argument, failing which a usage is printed, and the program exits.

In line 14, we make our first API call. We get the L-AST of the file referenced by

```
1  #!/usr/bin/env python
2  from redhawk.common.selector import S
3  import redhawk.common.get_ast as G
4  import redhawk.common.format_position as F
5
6  import sys
7
8  try:
9    filename = sys.argv[1]
10 except IndexError, e:
11   print "Usage: %s FILE"%(sys.argv[0])
12   sys.exit(1)
13
14 last = G.GetLAST(filename, None)
15
16 def HasBadName(f):
17   return "_" in f.name or not('A' <= f.name[0] <= 'Z')
18
19 bad_function_selector = S(
20     node_type='DefineFunction',
21     function = HasBadName)
22
23 for f in bad_function_selector(last):
24   F.PrintContextInFile(f, context=1)
```

Figure 5.8: Listing of an example *lint* program using the Redhawk API.

`filename`. We set database to None, to tell `GetLAST` to simply re-parse the file[10]. We have already seen lines 16 – 21 in the subsection on Selector API above. In line 23 we apply the `bad_function_selector` on the last to get an iterator to a list of results (functions that do not satisfy our coding standard). The `for` loop iterates over each of these results assigning the variable `f` to the current result. Line 24 prints the context of the node in the file (with one line above and below as the context).

Running the lint program on `stats.c` (Figure 5.7) gives the following output:

```
examples/stats.c:34:

examples/stats.c:35: > int main()

examples/stats.c:36:   {
```

[10]We are dealing with only one file here. The performance hit is not much. Also we want to keep the example simple — without keys, the location of the database, and such.

5.2.6 Parallelisation

Redhawk is pretty fast on large code bases for the kind of search it does. For example, it takes 34 seconds to query for function definitions anywhere in the entire source of `Django-1.2.1`, which contains 2273 *Python* files. However, a tool like `grep` takes only 2 seconds to search for the keyword `def`[11].To make Redhawk truly useful, it has to be fast enough to be a viable everyday-alternative to `grep`.

A crucial realisation is that querying across multiple files is an inherently parallel operation, since files can be queried independently. Redhawk supports parallel querying across multiple cores using the Parallel Python[Vanovschi, Retrieved 2011] framework. We show the effect of parallelisation in Table 5.10. As can be seen, we get performance improvements almost proportional to the number of cores.[12]

Table 5.10: Effect of Parallelisation on `redhawk query '**/DefineFunction'` on the source code of `Django-1.2.1`‘

Processor	Cores	Query Time (in seconds)	
		Without parallelisation	With parallelisation
Intel Core 2 6300 1.86GHz	2	34.9	18.6
Intel Xeon L5520 2.27GHz	4	33.8	10.9

The files are grouped into chunks, and the various chunks are sent as tasks to worker processes. The dependencies of each task also have to be kept track and sent along with the tasks for effective parallelisation. Choice of chunk sizes, and the number of worker processes affect the degree of parallelisation. The default

[11]It is not very fair to compare both tools, since Redhawk has 100% accuracy, and `grep` only 67.1% accuracy.

[12]Each data point present was an average of three runs under a normal processor load — browser, editor, and terminal.

values[13] have been chosen with some experimentation.

The Parallel Python module is loaded lazily so as to only make it an optional dependency. The implementation of the query task is in `redhawk.scripts.tasks`. The `TaskRunner` class in `redhawk.utils.task_runner` is responsible for running the task in parallel via the Parallel Python framework.

5.3 Summary

In this chapter, we first saw two ways to display the L-AST — one similar to scheme, and the other a pictorial representation. We then dealt with a particularly long section on Redhawk's query mechanisms; we looked at two kinds of query mechanisms with examples — the Redhawk XPATH Query mechanism, and the Selector API.

Each query mechanism has its advantages. In particular, the XPATH query mechanism is succinct, easy to understand, and suffices for most types of queries. The Selector API on the other hand is slightly verbose, and requires knowledge of the *Python* programming language. On the flip side, it is more powerful, and the intermediate results can be combined in any way required.

We then looked at the *Python* API that Redhawk exposes. We saw the implementation of a simplistic lint program using the API in 24 lines of code. Finally, we looked at Redhawk's ability to run queries in parallel. We observed speedups proportional to the number of cores. With multi-core processors, the Redhawk query system is within reasonable range of the (much more) naive `grep` search.

[13]The default number of worker processes is equal to the number of cores on the system. The default size of each chunk is 80 files.

CHAPTER 6

Conclusions and scope for future work

In this report, we proposed and implemented Redhawk, a standalone source code navigation system based on the concept of a language agnostic abstract syntax tree (L-AST). We chose *C* & Python to be proof of concept languages, due to the differences in their constructs, paradigms and typing systems. We proposed two query mechanisms on the resulting L-ASTs — one similar to the XPATH query language, and the other similar in spirit to the CSS3 Selectors. Each of these query mechanisms serve a different purpose. Several examples involving them were discussed. Furthermore, the process of querying was observed to be inherently parallel across files. This parallelism was exploited, enabling Redhawk to effectively parallelise itself across multiple cores. Finally, the standalone nature of the Redhawk Navigation System means that it can be used by other editors and IDEs. A Vim plugin is shown in Appendix B.

The use of the L-AST as a language agnostic representation for the purpose of navigation has been by and large successful. This is due to restricting its use for navigation purposes (as opposed to say, executing it).

6.1 Cultural Challenges

As with any new tool, there are a few cultural challenges:

- *Adoption*: Adoption of a new tool, that too, based on a radical idea like the L-AST will be slow. In fact, in the early stages of development, the author had to retrain himself to use Redhawk for development rather than `grep`.
- *Speed vs Accuracy*: Most simple tools that programmers use are fast but with low accuracy. The `grep` program is a classic example. The Redhawk navigation system is a slow, but highly accurate tool. This speed vs accuracy tradeoff will take some getting used to.
- *Learning Curve*: Redhawk's query mechanism is not as intuitive as `grep`. Redhawk performs a syntax based search on the source code. The programmer must know how to specify his queries using the Redhawk XPATH Query language or the Selector API. This learning curve, though not steep, exists, and will be a cultural challenge for the programmers.

6.2 Scope for Future Work

- The Redhawk Navigation system supports parallelisation across multiple cores using the parallel python library and is able to handle moderately sized code bases (~2200 files). Parallelisation across multiple servers is not yet supported and will be a useful feature for very large projects, like the source code of the Linux Kernel. This would involve shifting from the `shelve` data store to a robust, distributed key-value store with support for shards and replication.
- Currently, Redhawk only supports two languages — C & Python. Support for more languages will be essential for its success in the long run.
- *Fast paths* can be provided for some frequently occurring queries, like finding function definitions, function calls, and variable uses. A separate index can be built containing this information during parse time, alongside the L-AST index. This separate index can resemble a tag database, and answer these frequent queries. For other more complex queries, Redhawk can fall back to the standard L-AST index.
- Redhawk can become a full fledged *refactoring* tool by adding support to convert the L-AST back into the original language. This will allow programmers to directly edit the structure of the L-AST for refactoring.

APPENDIX A

Installing and Using Redhawk

A.1 Installing

`pip` is the recommended tool to install Redhawk. It goes by `python-pip` on debian/ubuntu and `pip` on the Python Package Index. The command "`pip install redhawk`" should install Redhawk, along with its dependency — `pycparser`. It is however recommended that the optional dependencies are also installed:

```
$ sudo easy_install pygraphviz
$ sudo pip install nose 'PyYAML>=3.09' 'nose>=0.11'
```

You could also use your distribution's package manager. On Ubuntu Lucid:

```
$ sudo aptitude install python-pygraphviz python-yaml python-nose
```

A.1.1 Dependencies

Runtime Dependencies:

- `pycparser` is required to parse C code into ASTs. This in-turn depends on Python-PLY (`python-ply` on debian-ubuntu).

Optional but highly recommended Dependencies:

- `pp` - Parallel Python is required for running queries in parallel. This speeds up queries by more than 2x. This is highly recommended if you are going to query large projects. The whole of Django can be queried in less than 20 seconds, by using parallel python (passing `-p` to the `query` command).

- `Python Graphviz` is required for generating pretty AST graphs. This package is an *optional* dependency, but highly recommended. This package goes by the name `python-pygraphviz` on Ubuntu, and depends on `graphviz`, and `dot`. (`Pip` seems to have a hard time install pygraphviz. Either `easy_install` or installing from your distribution's package manager should work).

Note: Run `build_tables.py` in the pycparser directory, to pre-generate the lex and yacc tables. This will enable quicker parsing of C files.

A.2 Using Redhawk

The `redhawk` program supports eight commands:

Command	Purpose
add	Add files to an AST index.
init	Create an EMPTY AST index.
listfiles	List all the files in the AST index.
prompt	Drop into a python prompt with helpful functions for exploring the parse tree.
query	Query for a pattern in a list of files, or in the index.
remove	Remove files from the AST index.
show	Show (visualize) a file either as text, or as an image.
where	Print the location of the current redhawk index (if there is one).

Table A.1: The `redhawk` commands.

The simplest way to run `redhawk` is to simply use a `query` command on a file (or directory). The `query` command as described above takes a Redhawk XPATH. For example, to find all functions at the module level in `counter.py`, we may do:

```
$ redhawk query 'DefineFunction' counter.py
```

In the case that the set of files is large and is to be repeatedly queried, a `redhawk` Language Agnostic Tree (L-AST) database can be created using the `redhawk init`

command. Files in the project can be added to the database using the `redhawk add` command.

The `show` command helps visualise the internal L-AST structure used. The command `redhawk show file.c` will show the L-AST of `file.c` in a lisp/scheme like (sexp) syntax. A more descriptive helpful visualisation can be obtained using the `-i` (or `-e`) flags, which show graphs (generated using `graphviz` using the `python-graphviz` module).

The `prompt` command drops you into a prompt for exploring and querying the L-AST. This enables the use of selectors, a very powerful method for finding what you want.

A.2.1 Getting Help

Redhawk is very well documented. The `pydoc` command should throw up documentation for most python modules. For example, to get more information on `redhawk.common.selector`, you can do `pydoc redhawk.common.selector`. Also, a set of introductory screencasts on Redhawk can be found at `http://www.youtube.com/watch?v=azaXpahrxww`.

A.2.2 Development (Compile-time) Dependencies

- Python YAML is required for generating the AST classes in node.py from a simple configuration file. This goes by the name `python-yaml` on debian/ubuntu.
- We believe in unit testing. Redhawk has 233 of them. The `nosetests` module is required for running the test suite.

Redhawk uses `git` for version control.

APPENDIX B

The Vim Plugin

A vim plugin has been written for Redhawk, using its command line API. The plugin may be downloaded and extracted into the user's `~/.vim` folder:

Currently, querying via the `redhawk query` command and replacing via an editable quickfix window is supported. The command:

```
:Redhawk query <query> [FILE]s
```

runs the Redhawk query on each of the FILEs, and populates the quickfix window. To replace we would do:

```
:Redhawk replace <query> [FILE]s
```

This runs the Redhawk query on each of the FILEs, and creates an editable quickfix window (similar to the `GReplace` plugin). The following list of commands are supported:

- `Redhawk`: Query and replace from within Vim.
- `RedhawkAdd`: Add to quick fix list.
- `RedhawkArgs`: Add args as files to redhawk command.
- `RedhawkBuf`: Add files in buffers to redhawk command.

APPENDIX C

The XPATH Grammar

A valid `XPathQuery` must satisfy the following grammar:

```
XPathQuery = AtomicQuery ('/' AtomicQuery)*


AtomicQuery = LocationQuery
            | (LocationQuery)     # ChildNodeMatchQuery query


LocationQuery =   .
                | ..
                | *
                | **
                | NodeQuery



NodeQuery = identifier? @[identifier=string]* @{codeblock}? Position*
Position = [ CommaSepNumbers ]


CommaSepNumbers = number | number , CommaSepNumbers


Tokens:
  identifier = regex("[a-zA-Z_][a-zA-Z0-9_]*")
  string     = regex("'[^']*'|" + r'"[^"]*"')
  codeblock  = regex("@{[^}]*}")
  number     = regex("-?[0-9]+")
```

APPENDIX D

The Node Configuration File and the Node Listing

D.1 The Node Configuration File

The L-AST node definitions are generated from configuration files in `redhawk/common`. The `_ast_gen.py` program is used to generate the `redhawk.common.node` and `redhawk.common.types` modules from the node and type configuration files (`_node_cfg.yaml` and `_types_cfg.yaml` respectively).

The YAML markup is used in the node configuration files. The following attributes are allowed.

Attribute	Default Value	Type
sexp		Yaml list
super	Node	Single class to inherit from
docstring	″	Single line doc string
args	″	CS-List
children	″	CS-List
xml	″	Yaml list
json	″	Yaml list
dot	″	Yaml list
optargs	″	CS-List

Table D.1: A description of YAML Markup attributes

A CS-List is a comma separated list like as follows:

```
This, is, a, comma, separated, list
```

As an example:

```
Return:
  docstring: "Represents a Return Statement."
  args: position, return_expression
  children: return_expression
  sexp: ["return", return_expression]
```

defines a `Return` node with the given docstring. It takes two arguments — `position` and the expression to be returned: `return_expression`. Its children list has only one element — the `return_expression`. Further the s-expression dictates how it is printed. If the `return_expression` were the expression 5, this would be printed as:

```
(return 5)
```

The `sexp` and `children` attributes can also take expand arguments using `*`. For example:

```
Module:
  docstring : "Represents a file or module."
  args: position, filename, children
  children: '*children'
  sexp: ["define-module", filename, children]
```

The children list of the `Module` node is the `children` attribute itself and not a one-element list containing the `children` attribute.

D.2 Node Listing

We now list all the LAST nodes. Optional attributes have a `?` after them.

Table D.2: Listing of all L-AST nodes and their attributes

Node	Attributes	Inherits From
Array	`array_type`	Type
Assert	`position`, `test_expression`, `message?`	Node
Assignment	`position`, `lvalue`, `rvalue`	Node
BaseType	`base_type`	Type
Break	`position`	ControlFlowStatement
CallFunction	`position`, `function`, `arguments`	Node
CaseDefault	`position`, `condition?`	ControlFlowStatement
Compound	`position`, `compound_items`	Node
Comprehension	`position`, `expr`, `generators`, `type`	Node
Constant	`position`, `value`, `type?`	Node
ContextVariables	`position`, `names`, `context`	Node
Continue	`position`	ControlFlowStatement
DeclareFunction	`position`, `name`, `arguments`, `return_type?`, `storage?`, `quals?`	Node
DeclareSymbol	`position`, `name`, `value?`	Node
DefineClass	`position`, `name`, `inherits`, `body`	Node
DefineFunction	`position`, `name`, `arguments`, `body`, `return_type?`, `storage?`, `quals?`	Node

Continued on next page

Table D.2 – continued from previous page

Node	Attributes	Inherits From
DefineType	position, name, type	Node
DefineVariable	position, name, init?, type?, quals?, storage?	Node
Delete	position, targets	Node
Dict	position, keys, values	Node
Enumerator	position, name, values	Node
EnumeratorType	enumerator_type	Type
ExceptionHandler	position, body, name?, type?	ExceptionsStatement
Exec	position, body, globals?, locals?	Node
Expression	position, operator, children	Node
Finally	position, body, final_body	ExceptionsStatement
For	position, init, condition, step, body	ControlFlowStatement
ForEach	position, target, iter_expression, body	ControlFlowStatement
FunctionArguments	position, arguments, var_arguments?, kwd_arguments?	Node
FunctionDecorator	position, decorator, function	Node
Generator	position, target, generator, condition?	Node
Goto	position, location	ControlFlowStatement
IfElse	position, condition, if_true, if_false?	ControlFlowStatement
Import	position, import_aliases	Node
ImportFrom	position, module, import_aliases	Import
Lambda	position, arguments, value	Node
Let	position, defvars, body	Node
List	position, values	Node
Module	position, filename, children	Node
ModuleAlias	position, name, asmodule?	Node
Pass	position	ControlFlowStatement

Continued on next page

Table D.2 – continued from previous page

Node	Attributes	Inherits From
Pointer	`ptr_type`	Type
Print	`position`, `values`, `stream?`	Node
Raise	`position`, `exception_type`	ExceptionsStatement
ReferVariable	`position`, `name`	Node
Return	`position`, `return_expression`	Node
Show	`position`, `value`	Node
Slice	`position`, `lower?`, `upper?`, `step?`	Node
SourceLabel	`position`, `name`, `statements`	Node
Structure	`position`, `name`, `members`, `storage?`, `quals?`	Node
StructureType	`structure_type`	Type
Switch	`position`, `switch_on`, `body`	ControlFlowStatement
TryCatch	`position`, `body`, `exception_handlers`, `orelse`	ExceptionsStatement
Tuple	`position`, `members`	Node
Union	`position`, `name`, `members`	Node
UnionType	`union_type`	Type
While	`position`, `condition`, `body`, `do_while?`	ControlFlowStatement
Yield	`position`, `yield_expression`	Node

REFERENCES

Abelson, H., **G. Sussman**, **J. Sussman**, and **A. Perlis**, *Structure and interpretation of computer programs*. Mit Press Cambridge, MA, 1996.

Bendersky, E. (2007). Xpath support in ElementTree. http://effbot.org/zone/element-xpath.htm.

Çelik, T., **E. Etemad**, **D. Glazman**, **I. Hickson**, **P. Linss**, and **J. Williams** (2009). Selectors level 3. *World Wide Web Consortium, Proposed Recommendation PR-css3-selectors-20091215*.

Clark, J., **S. DeRose**, *et al.* (1999). Xml path language (xpath) version 1.0. *W3C recommendation*, **16**, 1999.

Donzeau-Gouge, V., **G. Huet**, **B. Lang**, and **G. Kahn** (1980). Programming environments based on structured editors: The mentor experience.

Eastlake, D. and **P. Jones** (2001). US secure hash algorithm 1 (SHA1). Technical report, RFC 3174, September.

Eclipse Foundation (Retrieved 2011). The Eclipse IDE. http://eclipse.org/.

Hiebert, D. (Retrieved 2011). The Exuberant Ctags homepage. http://ctags.sourceforge.net/.

Hutton, G. and **E. Meijer** (1996). Monadic parser combinators. *Journal of Functional Programming*, **8**(4), 437–444.

Ierusalimschy, R., **L. De Figueiredo**, and **W. Filho** (1996). Lua-an extensible extension language. *Software Practice and Experience*, **26**(6), 635–652.

Jason Merril (2003). GENERIC and GIMPLE: A New Tree Representation for Entire Functions. *GCC Developers Summit*.

JetBrains, I. (Retrieved 2011). Intellij Idea. www.intellij.com.

Kernighan, B., **D. Ritchie**, and **P. Ejeklint**, *The C programming language*, volume 78. Citeseer, 1988.

Meertens, L., **S. Pemberton**, and **G. van Rossum**, *The ABC structure editor: structure-based editing for the ABC programming environment*. Citeseer, 1992.

Perlis, A. (1982). Special feature: Epigrams on programming. *ACM SIGPLAN Notices*, **17**(9), 7–13.

Steffen, J. *et al.*, Interactive examination of a C program with CScope. *In Winter USENIX Technical Conference*. 1985.

van Rossum, G. and **F. Drake Jr** (Retrieved 2011). The Python Tutorial: Release 2.6.6. More on defining functions. http://docs.python.org/release/2.6.6/tutorial/controlflow.html#more-on-defining-functions.

Vanovschi, V. (Retrieved 2011). Parallel python software. http://www.parallelpython.com.

Yegge, S. (2010). Scalable programming language analysis. http://vimeo.com/16069687.