ENGG4801 Project Proposal

J. D. Bartlett, 40791032

2 April 2009

Contents

1 Project overview **1**
1.1 Aims and coverage . . . 1
1.2 Motivation . . . 1
1.3 Example cases . . . 2
1.3.1 Example: Python extension modules . . . 2
1.3.2 Example: Parser construction . . . 2
1.3.3 Example: Unit and dimension checking . . . 3
1.3.4 Other assorted examples . . . 3

2 Literature review **4**
2.1 Extensible programming . . . 4
2.2 Projects with similar aims . . . 4
2.2.1 PyPy . . . 4
2.2.2 Psyco . . . 5
2.2.3 Pyrex and Cython . . . 6
2.2.4 Inlining C code in Python . . . 6
2.3 Parser generator technologies . . . 6
2.3.1 mxTextTools . . . 7
2.3.2 SimpleParse . . . 7
2.3.3 PLY . . . 7
2.3.4 ANTLR . . . 7
2.3.5 SPARK . . . 8
2.3.6 Other Tools . . . 8
2.4 Graph manipulation technologies . . . 8
2.5 Code generation technologies . . . 9

3 Project details **10**
3.1 System overview . . . 10
3.2 Writing source code . . . 10
3.3 Using Python code . . . 11
3.3.1 Category system . . . 11
3.3.2 Interface with Python . . . 12
3.3.3 Tree manipulation . . . 16
3.4 Invoking the compiler . . . 17
3.5 Extending the compiler back-end . . . 18

4 Plan for completion **18**
4.1 Current progress . . . 18
4.1.1 Investigation of technologies . . . 18
4.1.2 Language definition . . . 18
4.1.3 Tree manipulation library . . . 18
4.2 Milestones . . . 19
4.3 Timeline for milestone completion . . . 19
4.4 Risk assessment . . . 20

References **20**

1 Project overview

1.1 Aims and coverage

This project aims to design and implement software. The following points describe the envisioned software:

- The intended users of the software are computer programmers.
- The software will act as a compiler taking as input source code written using a well-defined syntax, and generating machine code as output.
- The software will provide a framework through which the user can customise the compilation process by providing Python files which specify some part of the compiler behaviour.
- The software will be platform- and system-independent to the greatest degree possible.

1.2 Motivation

The overall motivation of this project comes from the following two premises:

- Programmers should have easier ways of generating machine code than those currently readily available.
- Compiler preprocessors shouldn't have to be as restrictive as they are.

This project stems from a consideration of common existing programming languages. All software projects may be divided broadly into two categories: those for which compiling the software to native machine code is an important requirement; and those for which it is not. Projects may have such a requirement, for instance, due to speed constraints, or because the software must be deployed onto a microcontroller as part of an embedded system.

Many modern interpreted programming languages make use of features which can provide great flexibility to programmers, but which make it difficult to implement a compiler for those languages. These languages are often prime choices for use by projects where compiling to native machine code is not a requirement. Without the ability to compile code, such programming languages cannot be directly used on projects where generating machine code is important, but there is no reason why the flexibility of such languages could not be used to make it easier to develop software to be compiled.

Traditional compiled programming languages provide numerous preprocessor directives to enable the user to in some way control the behaviour of the compiler from within the source code. This project aims to investigate and implement a radically new approach to compiler preprocessors: rather than having a restrictive set of possible preprocessor directives, this project intends to allow use of the full scope of the high-level dynamic programming language Python to control the compilation process.

Because users will be provided with more control over the compilation process, they will have the ability to express higher-level concepts directly in the source code, provided that they also give instructions to the compiler as to how

to interpret those concepts. It is anticipated that by giving users the ability to express higher-level concepts in the source code, length of code will be reduced and code maintainability and readability will be improved compared with existing compiled languages.

Along with the ability to express higher-level concepts will come the ability to express domain-specific concepts which, using existing compiled languages, would be difficult to express, or would require lots of repetitive code to express. The software proposed by this project will give users the ability to express such concepts simply.

Developing the software will also have the side effect of providing an interface by which Python programmers can straightforwardly generate machine code, simplifying the task of writing portable compilers for general or domain-specific programming languages in future.

1.3 Example cases

Described below are a few specific examples of situations in which the software proposed by this project may be usefully employed. For a more detailed look at the intended behaviour of the software, see Section 3.

1.3.1 Example: Python extension modules

The Python programming language provides a mechanism for writing extension modules in C. In order to write such a module, a programmer is required to use macros to build Python objects from C primitives, use macros to construct Python exceptions where necessary, correctly deal with Python exceptions from called functions, increment and decrement reference counters on Python objects as appropriate, and include a special methods table to tell Python which functions should be available from Python code and how.

These important tasks must be done in every extension module in order for the module to work correctly. If the proposed software system were used to write Python extension modules, the user could write a single Python file which defines certain keywords and how the compiler should treat them. The user could then set about writing source code which imports the keywords from the Python file and makes use of them. The compiler would follow the customisations defined by the keywords, and in doing so would automatically construct the features common to every extension module.

1.3.2 Example: Parser construction

There are many situations in programming in which a programmer would want a program to use a well-defined grammar to parse text and form a syntax tree representing that text. Such situations may occur in software compilers and interpreters, but programmers may also want software to read so-called little languages for various purposes. There are numerous available compiler generators (or more correctly parser generators) which take a grammar definition and produce source code which parses text according to the given grammar. Using such tools avoids a programmer having to write a parser from scratch.

As an alternative to making use of an existing parser generator, a user could choose to tackle the problem from a different direction using the software pro-

posed in this document. The user could write a Python file telling the compiler to interpret a grammar definition as a parser for that grammar. If this Python file were then made widely available, a user who wished to write a compiler could write source code which imported this file, and in the same source file could define the parser by specifying the grammar definition, as well as writing the other behaviour of the compiler such as semantic analysis and code generation.

1.3.3 Example: Unit and dimension checking

In contexts such as science and engineering, variables may represent physical quantities. In such cases, it is vital that units be handled correctly. For instance, a variable that contains a measurement in miles should never be assigned to a variable that contains a measurement in metres without being multiplied by the appropriate conversion factor. Similarly, a value with one dimension (such as length) should never be stored in a variable which is designed to contain values of a different dimension (such as time).

The software proposed by this project would make it possible to write a Python file to extend the compiler, enabling unit checking of variables. The Python file could, for instance, define a keyword which allowed units to be defined in software source code. Within the keyword definition, a compiler customisation could be specified for cases when defined units were used in variable declarations. After writing such a file, programmers would be able to import the keyword for unit definition, and use it to define units and make use of them in their source code. When the code is compiled, the compiler would call the functions defined in the Python file, which would check that all operations are dimensionally valid, and report errors for those which are not.

The idea of incorporating unit checking into compilers has been investigated by many people (see, for instance, [1, 2, 3, 4, 5]). This example is included here not as a novel invention unique to this project, but as an example of a feature which is not included in the software but could easily be added by a user. This example is investigated in more detail in Section 3.3.2.

1.3.4 Other assorted examples

After such examples it should scarcely be necessary to point out that the software proposed will have the flexibility to implement many of the simpler tasks made available in other programming languages. For instance, implementing macros or template functions would be quite possible. It would also be possible to define specially-behaving data types such as enumerated types, subsets of numerical types, or imaginary numbers (assuming that such types were not defined in the base language). Even more specialised concepts such as vectors, matrices and tensors could also be defined.

With regard to implementing macros in the language, it is worth noting that the proposed behaviour of the software is such that defined macros would be easily constrained so that they may only be inserted in certain contexts within the syntax tree, avoiding many of the potential mistakes which become possible in languages such as C where preprocessor macros are substituted at the level of tokens or text.

2 Literature review

2.1 Extensible programming

In the 1960s and early 1970s, much work was done on the concept of *extensible programming*. The concept revolved around the idea of providing mechanisms by which the core features of a programming language could be supplemented, often by making use of some kind of *meta-language* in which the definition of the base language was expressed. This work on extensible programming often focussed particularly on the abilities to define macros and to adapt the grammar of the language. For a 1975 review of the topic of extensible programming, see [6].

More recently, there has been renewed interest in the concept of extensible programming. One advocate of this concept is Gregory Wilson of the University of Toronto, who argues that next-generation programming languages should have the ability to be customised using plug-ins, should allow programmers to extend their syntax, and should store programs as XML documents so that data and meta-data can be represented and processed in a uniform way. Wilson claims that “these innovations will likely change programming as profoundly as structured languages did in the 1970s, objects in the 1980s, and components and reflection in the 1990s.” [7]

While it is important to know of other work in areas related to the current project, it should be noted that the software proposed in this document falls short of Wilson’s ideal, and even lies outside the historical realm of extensible programming. The aims of the current project do overlap with Wilson’s ideas in the goal of constructing a compiler which can be customised using what could be referred to as plug-ins; this project does not, however, make any attempt to allow programmers to extend the language syntax. This is a deliberate choice, based on the idea that a programming language should make it at least as easy to write readable, maintainable code than any other sort of code. Redefinable syntax leaves programmers with too great an ability to construct unreadable programs.

2.2 Projects with similar aims

The concept of taking the flexibility that is available in many interpreted programming languages and making it available to developers of compiled software is not a new one. There are a number of prominent projects which already attempt to achieve this in various ways.

2.2.1 PyPy

The PyPy project [8] is centred around the primary goal of implementing a viable version of Python in Python itself.

> The PyPy project seeks to prove both on a research and a practical level the feasibility of constructing a virtual machine (VM) for a dynamic language in a dynamic language—in this case, Python. The aim is to translate (i.e. compile) the VM to arbitrary target environments, ranging in level from C/Posix to Smalltalk/Squeak via

> Java and CLI/.NET, while still being of reasonable efficiency within these environments. [9]

The PyPy virtual machine is written using a subset of Python (referred to as restricted Python, or RPython). The PyPy project has then written a toolchain which may be used to translate the VM to some target environment. Commonly the VM is translated to C and then compiled. When tested against performance benchmarks, the compiled PyPy VM typically performs each iteration in between three and ten times the time taken by the standard distribution of Python, which is implemented in C [9].

The PyPy toolchain can also be used to compile arbitrary programs from RPython to C or some other target language. In theory this allows programmers to use the features of the Python programming language, and to end up with machine code. In practice, there are drawbacks to this approach. One important obstacle faced by many newcomers to PyPy is that PyPy does not translate the whole range of the Python programming language, but only the restricted subset designated RPython by the PyPy project. RPython is not clearly defined or specified. In fact, the only detailed definition of what is and is not allowed in RPython is the implementation. It is certainly an obstacle to programming for programmers to be unsure of whether certain code is or is not allowed in the programming language until they try to compile it.

One key concept in the PyPy project is the distinction between the code that is being compiled and code that will be executed at compile-time as part of the translation process. In PyPy, both these categories of code are written in Python (and some code may even fall into both categories), but the code that is to be compiled must be written using RPython. There is no such restriction on the code which is executed at compile-time as part of the translation tool-chain, which may be written using the full range of features available in the Python language.

2.2.2 Psyco

Psyco [10] is a Python extension module which is designed to speed up the execution of Python code. Psyco is based on the concept of just-in-time (JIT) compiling, but might better be thought of as a just-in-time specialiser [11]. At run-time, it infers restrictions on variables from the values that a Python program manipulates. It then emits efficient machine code for the functions based on those restrictions. If data comes along later which does not match the inferred restrictions, Psyco can emit new machine code. The program is optimised at run-time for the data that it is currently handling.

Psyco has the advantage that existing Python code does not have to be modified in order to use it with Psyco. A programmer simply needs to include the Psyco module and the program will run with the performance benefits.

Running common Python code with Psyco typically results in a speed approximately four times that achieved by interpreting the Python code without Psyco. The performance gain varies depending on the code being executed. In situations where many repetitions and manipulations are performed on data of a fixed type, Psyco typically results in higher performance gains, up to 10 or 100 times that achieved without Psyco [10].

The seem to be several drawbacks of Psyco. Firstly, it is only implemented

for Intel processors (although it does run independent of operating system). Secondly, it uses a lot of memory [10]. Another drawback is that complete native machine programs are not generated, so in order for customers to run software which uses Psyco, the customers must have Python installed on their computers.

2.2.3 Pyrex and Cython

Two interesting developments along a similar theme are the Pyrex project [12] and a fork of Pyrex known as Cython [13]. Pyrex author Greg Ewing sums up Pyrex by saying "Pyrex is Python with C data types" [14]. Pyrex starts with Python code which is annotated to restrict the possible types of certain variables, and generates C code. In cases where data types are specified, C data types are used. For variables whose types are not specified, Pyrex will generate the needed C code to construct Python objects. Since extension modules are linked against the Python executables, almost all valid Python code is valid code in Pyrex.

Cython is an incomplete project which is based on Pyrex. Cython has the same essential goals as Pyrex, but provides a number of additional features [15]. Both the Pyrex and Cython projects build extension modules for Python, and require that Python be installed on the target system in order that software be executed.

2.2.4 Inlining C code in Python

It is worth mentioning that there are a number of projects which have aims similar to those of this project in that they aim to improve the performance of high-level interpreted languages. In the case of Python, examples of software projects which allow some form of embedding of C code within Python include Cinpy [16], Weave [17] and PyInline [18]. The aims of these projects differ significantly from those of the current project in that they aim to improve the performance of Python code without generating complete machine code for programs. They are mentioned here in order to give a more complete overview of the work that others are doing in the same area.

It should be noted that the problem of improving the speed of high-level interpreted languages has attracted the attention of many. For instance, Google has recently announced the project *Unladen Swallow* which aims to develop an implementation of Python at least five times the speed of the current C implementation of Python [19, 20].

2.3 Parser generator technologies

The major part of this project will be writing a compiler. There are already numerous existing tools for the purpose of parsing input files according to a given grammar. In this section various such tools and resources relating to the Python language will be discussed.

The book *Text Processing in Python* [21] contains an excellent overview of the various tools available for processing text in Python. In particular, Chapter 4 talks about parsers and demonstrates use of the mxTextTools, SimpleParse and PLY libraries to parse marked-up text.

2.3.1 mxTextTools

The mxTextTools Python module [22, 23], provided as part of eGenix.com's open source mx Base Distribution, is a module designed to provide fast text processing and parsing capabilities to Python programs. The module provides an advanced form of finite state machine written in C. The machine is specialised for text processing, and is able to recursively call itself to construct parse trees from text input. Because the state machine is implemented in C, using the mxTextTools module to process text is, in general, far more efficient than writing Python code to do the same job.

To make use of the mxTextTools state machine, a programmer must provide the module with a low-level program for the state machine to execute [23]. This program must be provided as a set of tuples containing instructions and instruction parameters. It is relatively easy to construct the required set of tuples for a simple program such as a lexical analyser (or scanner), but to construct the tuples for a parser for an entire programming language would be like using nothing but assembly language in a large software project: doing so would be possible, but it would take a lot of time and the code would be very hard to maintain.

2.3.2 SimpleParse

SimpleParse [24] is a Python module which simplifies the process of writing parsers using the mxTextTools module. SimpleParse provides access to most of the features of mxTextTools, but provides a much nicer interface. It allows parsers to be defined using a variant of Extended Backus-Naur Form, and converts that internally into the set of tuples needed by mxTextTools. The outcome is that text parsing can be performed in Python with the high speed available through mxTextTools, but without the drawback of having to write low-level state machine code.

The key drawback that is obvious when using SimpleParse is that there is no built-in facility for syntax error recovery.

2.3.3 PLY

PLY [25] is a straightforward Python implementation of the lex and yacc parsing tools. It provides the ability to create Python source code for lexers and parsers based on language definitions. It includes support for precedence rules and error recovery.

PLY has the drawback that it generates pure Python code, and therefore parsers constructed using PLY run more slowly than those created with SimpleParse or mxTextTools.

2.3.4 ANTLR

ANTLR [26] (ANother Tool for Language Recognition) provides a framework for generating parsers and other language tools. From a formal language grammar, it generates parser code in any of a variety of target languages, including Java, Python and Ruby. It is designed around the principle of generating human-readable parser code. Along with its text-parsing tools, it provides tools

for applying formal grammars to tree structures. ANTLR has error handling described by its creator as "pretty flexible and decent" [27].

As with PLY, ANTLR generates pure Python code and therefore its parsers run slowly compared with other tools. ANTLR provides somewhat more powerful parsing facilities than PLY. It has a large following, is under active development, and is well-documented to the point of having a published reference book [28].

2.3.5 SPARK

SPARK is included here because it seems to be a well-regarded and widely-used parsing tool. SPARK [29, 30, 31] is a framework for writing little languages in Python. The framework includes facilities for performing semantic analysis and code generation. The framework makes use of a technique known as Earley parsing, which is able to handle any context-free grammar [31]. SPARK, unlike PLY or ANTLR, does not generate Python code to do the parsing. Instead, the parsing algorithm interprets the grammar at run-time.

The approach which SPARK uses, while very powerful, has the disadvantage that it is also somewhat slow. A parser written using SPARK would typically run more slowly than one generated by PLY or ANTLR.

2.3.6 Other Tools

There are a myriad of text-parsing tools available on the Internet. A useful resource is Ned Batchelder's webpage [32], which contains a list of over thirty different Python parsing tools together with a brief description of each.

2.4 Graph manipulation technologies

The software proposed in this document will provide users with the ability to customise the process of creating an output Abstract Syntax Tree (AST) from the parse tree. Many users will therefore need the ability to easily compare and manipulate tree structures in Python. Such a facility will also be useful for the development of the compiler software itself.

It should be noted that while it is common to talk about trees in software compilers, there is no reason why a so-called Abstract Syntax Tree should be a tree rather than a more general directed acyclic graph (DAG). For an overview of the theory of graph transformation and its applications in programming, see [33].

There are not many tree pattern-matching or transformation routines easily available in Python. A thorough search showed only one promising candidate, Sam Wilmott's pattern-matching library [34]. This library provides facilities to build patterns for matching sequences of generic objects. One particularly appealing feature of the library is its use of overloading operators to make code for the construction of patterns visually simple.

The features of Wilmott's library are extensive. Although it would be possible to extend the library to cope with tree structures, many of the library's features would not be needed in the context of the software described in this document, and there would be many features not provided by the library which would be useful to have. Taking all things into consideration, it seems that the

most sensible approach is to build a new custom module for constructing trees and performing pattern-matching and manipulation of trees.

2.5 Code generation technologies

The final phase of any compiler is to generate from the compiler's internal representation of a program the output code for that program in some desired format.

The aim of this project is to build a compiler which ultimately results in native machine code across a wide variety of target systems and platforms. In order to achieve this goal, the software proposed by this project will make use of an existing, available technology to generate the output code. The obvious candidate is the GNU Compiler Collection (GCC) [35].

The GCC project can be roughly divided into front-ends and back-ends. Front-ends read source code and build an internal representation of that code. Back-ends take the internal representation and generate output code. Currently, GCC has front-ends for many languages, including C, C++, Objective C, FORTRAN, Java and Ada. It has back-ends for numerous different target architectures and operating systems.

The concept of using GCC to act as a back-end for the software proposed here can be divided into three distinct approaches: the software could generate code in a well-defined language supported by GCC (e.g. ANSI C-89) which could then be passed to GCC; the software could be implemented as a GCC front-end and link directly with the back-end; or the software could generate code in some well-defined file format which corresponds closely to the GCC internal representation and which could then be compiled by a custom GCC front-end built for such a purpose.

The second possibility—writing this project as a GCC front-end—may be ruled out from the outset. Doing so would introduce strong coupling with GCC which would not be ideal. Writing the system primarily in Python has various advantages, particularly when it comes to interfacing with the user's code.

The very task of writing a GCC front-end is not a particularly simple one. There is currently no way to compile a front-end as a stand-alone library which can be dynamically linked with a pre-installed copy of GCC. Instead, the front-end must be linked with the rest of GCC at build-time. There is some documentation and are some examples of how to go about writing a GCC front-end. For instance, in [36] the author discusses how the process of writing a GCC front-end for a functional language improves the language's efficiency by correctly handling tail-recursive calls.

There would be various advantages to the possibility of writing a minimal GCC front-end to convert an input file to GENERIC (a tree structure used by GCC to represent programs internally). Doing so would make it easier to implement GCC front-ends in languages other than C, which would make writing cross-system compilers a much more accessible endeavour.

At present, it seems that the possibility of generating intermediate C code will be more straightforward than that of writing a small GCC front-end to read tree files. The latter possibility seems like a prime candidate to be an enhancement for the language in future.

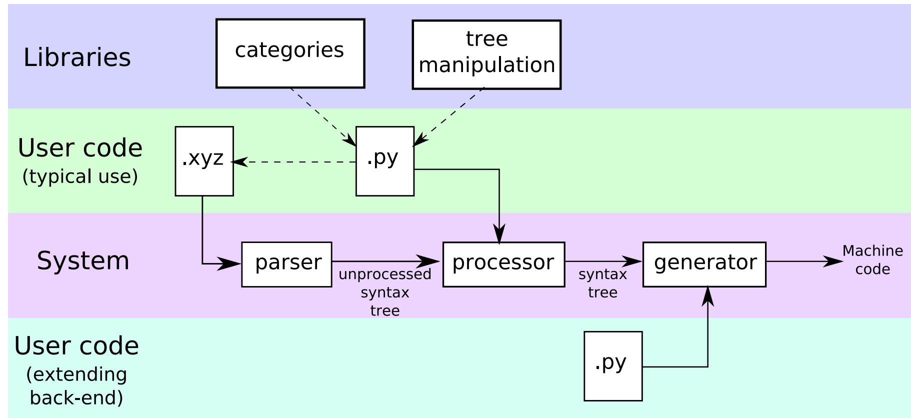

Figure 1: Overview of proposed system

3 Project details

3.1 System overview

Figure 1 shows a pictorial overview of the proposed system. In the figure, the second band from the top represents the code written by a user in a typical situation. The file denoted ".xyz" represents a source file to be compiled by the system. Section 3.2 introduces the syntax of such source files using a simple program as an example. Section 3.3 looks at how a user may introduce Python code to extend the way in which the compiler behaves when compiling the user's program. It also outlines the libraries which the system will make available for use from within such Python code. Section 3.4 shows how the compiler may be invoked, both from the command-line and from within other Python projects. Finally, Section 3.5 discusses how one could extend the compiler to control the kind of output generated, represented by the bottom-most band in Figure 1.

3.2 Writing source code

The software implements a syntax which is relatively simple, but is still somewhat different from the syntax of either C or Python. The purpose of the current section is to introduce this syntax. To do so, some simple example code demonstrating the basic features of the language is presented below. Please note that the syntax used in this example focuses on simplicity rather than efficiency. It is likely that in future some amount of "syntactic sugar" will be introduced to improve the usability of the software.

```
# Define a constant.
C: 42

min: function(a: int, b: int -> int)
    if a < b
        return a
    return b
```

```
sum_less: function(x: int -> int)
    # Returns the sum of all integers less than or equal
    # to x. If x is greater than C, C is used instead.

    result: int
    i: int

    x = min(x, C)
    result = 0
    i = 0
    while i <= x
        result = result + i
        i += 1
    return result

main: function()
    r: int

    r = read()
    write(sum_less(r))
```

There are a few points to note about this listing. It should be obvious to the reader that indentation is used to define the structure of the program. It is also worth noting that, although this programming language has not been implemented at the time of writing, it is clear from reading the source code what the program listed above is intended to do.

A key feature of the syntax shown above is the use of the colon symbol (:) to define symbols to the compiler. This notation will be very useful when it comes to extending the compiler functionality with Python code.

3.3 Using Python code

Without the ability to extend the compiler using Python, this project would be just another compiler. This section describes the basic interface that this compiler will have with Python, as well as some of the tools that users will have available to them within their Python code.

3.3.1 Category system

This project makes use of a small library which the author has previously written which allows Python objects to specify what behaviours they do and do not implement. This library is independent of the software as a whole, but the software makes use of the library as part of its interface with a user's Python code. In order for a user to extend the system with Python code, the user must be familiar with this library.

The library allows the construction of *categories*, whose meaning should be defined by human-readable specification comments. If an object claims to be a member of a category, it guarantees that it fulfils the behaviour specification for that category. The library provides mechanisms to guarantee that an object is a member of a category, and to test whether or not an object claims to be in a

particular category. This allows objects to say in a simple way what behaviours they implement, while leaving them free to implement those behaviours however they wish (i.e. without forcing them to inherit from a superclass which may provide implementation which they do not require).

The `categories` module provides the following functions and objects:

- `guarantee_membership(obj, category)`—guarantees that the specified object is a member of the given category.
- `remove_membership_guarantee(obj, category)`—this will remove any guarantee that has been made previously that the object is a member of the category.
- `ismember(obj, category)`—tests whether or not an object is a member of a particular category.
- `issubcategory(category1, category2)`—tests whether `category1` is or is not a subcategory of `category2`.
- `requiremember(obj, category, [message])`—tests whether the given object is a member of the given category, and raises `CategoryError` if not. If a message is specified it is used as the parameter of the exception.
- `CategoryError`—an exception type for use when a function requires that its parameter belongs to a certain category but that requirement is not met.
- `BehaviourError`—an exception type for use when an object is guaranteed to be in a certain category, but the object is behaving contrary to the definition of that category.
- `CATEGORY`—a category which all categories must be a member of. Full specification is available in the source code documentation.
- `Category(...)`—a class whose instances are always valid categories. Instances of this class may specify categories that this category implies and objects or categories which are known to fulfil this category's specification. It is also possible to specify which built-in classes are implementations of a category. The complete specification for this function is available in the source code documentation.
- `ITERABLE, CONTAINER, SIZED, MAPPING`—predefined categories. These have been set up so that the built-in objects and types which fulfil these categories behave correctly when `ismember()` is used.

3.3.2 Interface with Python

In order to convey a basic understanding of the framework used by the proposed software, consider as an example the situation described in Section 1.3.3—the user wishes to write a program which performs calculations on quantities with units, and wants the compiler to perform unit checking on variables and calculations. This section's treatment of this example is not complete; it provides only an outline in order to convey to the reader the interface which the proposed software provides to users.

In such a situation, the user may develop a notation which looks something like this:

```
from library import unit

# Define the units we will be using.
m: unit()
s: unit()
kg: unit()
N: unit(kg*m/s/s)

req_force: function(mass: float*kg, dist: float*m, ...
        time: float*s -> float*N)
    # Calculates the total force required to move
    # an object of a given mass by a given distance
    # in a given amount of time assuming a constant
    # acceleration.

    accel: float*m/s/s

    # s = 0t + 1/2 at^2
    accel = 2 * dist / time**2

    # F = ma
    return mass * accel
```

It is an important point that the user is not able to modify the *syntax* of the language. The syntax is fixed by the language specification, but is sufficiently flexible that there are things which are syntactically correct in the language but which have no meaning for most built-in keywords. For instance, the variable declaration `accel: float*m/s/s` is syntactically correct because the symbol definition syntax allows arbitrary expression forms on the right-hand side of the colon. If the user were to write `accel: int*float`, this would also be syntactically correct, but because `int` and `float` have no defined behaviour when multiplied in this context, this is a semantic error.

In order to make the source code given above behave correctly, the user must provide a Python module called `library` which contains the definition of the keyword `unit`. An example outline of such a definition is below.

```
class UnitKeyword(object):
    def __init__(self):
        guarantee_membership(self, NODE_ALLOWS_CALL)

    def node_call(self, parameter_nodes):
        ...
        return UnitNode(...)

unit = UnitKeyword()
```

Here `NODE_ALLOWS_CALL` is a category defined by the compiler.

In order to process a syntax tree into a form for which it can generate output code, the compiler will process each node of the syntax tree according to certain rules. When it gets to the line of the source code that says `m: unit()`, it will first process the expression `unit()`, then assign it to the symbol `m`. The compiler will look at the Python object that is represented by the symbol `unit`, and will ascertain that it implements behaviour allowing it to be used in the context of a call. The compiler will then call the function `unit.node_call()` and will pass to that function a list containing the tree nodes for the expressions appearing as parameters within the brackets. In the case of the expression `unit()`, this list will be empty. The function `unit.node_call()` returns a tree node which is used to replace the call node in the compiler's syntax tree.

Once the behaviour of the `unit` keyword has been defined to construct a `UnitNode` object, the user would proceed to defining what a `UnitNode` does. An outline of this definition is provided here.

```
class UnitNode(object):
    'Represents a unit or combination of units.'

    def __init__(self, ...):
        ...
        guarantee_membership(self, NODE_ALLOWS_MUL)
        guarantee_membership(self, NODE_ALLOWS_DIV)

    def node_mul(self, other):
        ...
            # In the case of composite units:
            return UnitNode(...)
        ...
            # For type/unit combinations, such as
            # int * metres or int * metres / seconds
            return UnitTypeNode(...)
        ...
        # You would also need to deal with the cases
        # of unit * number and unit * variable.
    ...
```

Similar to the previous code segment, the `UnitNode` objects specify that they may be used in the context of multiplication and division. When actually used in multiplication or division, a `UnitNode` will return another `UnitNode` or a `UnitTypeNode` which the compiler will then use to replace the binary operation node. Finally, the user would have to define the `UnitTypeNode` class, which would perform the actual type checking.

```
class UnitTypeNode(object):
    'Represents unit/type combination.'

    def __init__(self, ...):
        ...
        guarantee_membership(self, NODE_ALLOWS_ASTYPE)
        guarantee_membership(self, NODE_ALLOWS_TYPECHECK)
```

```
def node_astype(self):
    ...
    # Return the type to actually use in memory.
    return self.baseType

def node_typecheck(self, expr):
    ...
    # Perform type checking for assignment here.
    return isCompatible
```

This class specifies that its instances may be used in the same context that types would normally be used. It tells the compiler what type to actually use internally to represent variables of this type, and provides a mechanism for checking whether or not a particular expression may be assigned to a variable of this type.

The compiler will allow the user to make use of Python-defined objects in virtually any context that is syntactically correct. The following table describes each of these contexts.

Context	Example	Description
Symbol definition	`x: keyword`	Used to define a symbol to the compiler. For instance, `x: int` defines x to be a variable of type int.

Multiline symbol definition	`x: keyword`
	`    line`
	`    line`
		Also used to define a symbol to the compiler, but makes use of an indented block of lines to further define the symbol. This is the same syntactic form as used to define functions in the examples above.

Statement block	`keyword expr`
	`    line`
	`    line`
		Use of an indented block to do something other than defining a symbol. A common use of this context would be to define a program control structure.

Binary operation	`keyword + 3`	For each of the binary operators defined by the language, there is opportunity for a Python-defined object to customise its behaviour when used as the left or the right operand. If the left operand has a valid customisation, this is given precedence over the right operand's customisation.
Unary operation	`-keyword`	Python-defined objects have the opportunity to customise their behaviour under each of the unary operators defined in the language.

Context	Example	Description
Call	`keyword(args)`	Any context where an expression is followed by parentheses, optionally enclosing a list of parameter expressions.
Attribute reference	`keyword.attr`	Used to access attributes of an object or struct.
Subscription	`keyword[index]`	Used, for instance, to access elements of an array by index.
Assignment	`keyword = expr`	At attempt is made to assign to the object. It is uncommon for this to be required by Python-defined objects (usually users would only assign to variables).
Use as type	(Must be inserted programmatically. Typically used in conjunction with the symbol definition context.)	Specifies that entries may be made in the symbol table for variables which have this node as their type. Must specify what type to actually use in memory.

Instance operations	`x: keyword`
	`x + 3`
	`x = 17`
	`x[9]`
		A Python-defined object that specifies that it may be used as a type may also specify the way in which variables which make use of this type behave in each of the contexts defined in this table.

The compiler will also provide users with the ability to do the following from Python code:

- Perform default processing on tree nodes—so that a custom tree node may tell the compiler to process its child nodes.
- Raise exceptions which translate to compiler errors.
- Match and manipulate compiler trees as described in the following section.

3.3.3 Tree manipulation

In order to improve the ease with which the compiler tree can be processed into a form suited for output, the proposed software will provide a library of Python functions to perform pattern-matching and manipulation on compiler trees (recall that trees need not technically be trees, but must be DAGs).

In this section, a *pattern* is a tree made up of both concrete nodes and nodes which may match tree nodes or subtrees based on a particular set of criteria.

The tree manipulation library will provide the following definitions:

- A category defining the behaviour of a tree node.
- `traverse(node, [pre], [post])`—a function which performs a generic traversal from a starting tree node executing the `pre()` function on each

node before visiting that node's children, and the `post()` function afterwards. Along with this function, the library will provide exceptions which can be raised to indicate that tree traversal should terminate, or that the current node's children should not be traversed.

- `matchtrees(target, pattern)`—performs pattern-matching on the target tree using the specified pattern. Returns a mapping from template nodes within the pattern to the actual nodes corresponding to those templates in the target tree.
- `substitute(node, mapping)`—performs substitution on the given tree by replacing template nodes in the tree with nodes specified according to a mapping.
- `copy_substitute(node, mapping)`—this function has the same functionality as `substitute()`, but rather than performing the substitution in-place, constructs a new tree that results from the substitution.
- `findonce(tree, pattern)`—performs a traversal of the given tree searching for the first subtree that matches the specified pattern.
- `findall(tree, pattern)`—performs a traversal of the given tree and finds all non-overlapping subtrees which match the specified pattern.
- `replaceall(tree, find_pattern, replace_pattern)`—when this function is called, all non-overlapping subtrees of the given tree which match `find_pattern` will be found and replaced with copies of `replace_pattern` which have had any template nodes replaced by the subtrees that the same template nodes matched against `find_pattern`.
- `TreeNode`—a class which implements the tree node specifications.

3.4 Invoking the compiler

After having written code, the user must be able to compile it. There will be two ways to invoke the compiler: from the command line, and from Python code.

The most common usage of the compiler will be to invoke it from the command line. When run in this way, the compiler will take as input the name of the file to compile, and optionally the target for code generation. If no target is specified, the default will be to generate machine code for the current architecture (as discussed above, this may occur via intermediate C code). In future the compiler may include different targets, but developing code generators for such targets is beyond the scope of this project.

The software will also provide an interface for invoking the compiler from within a Python program. The following list describes the functions which will be made available as part of this interface.

- The `compiler()` function will create a compiler object for a specific code generation target (defaulting to machine code on the current system).
- If `comp` is a compiler object, `comp.feed()` will provide the compiler with a string which the compiler will parse and add to its syntax tree.

- `comp.addtree()` will add a tree to the compiler's syntax tree as a child of the program node.
- `comp.generate()` will process the compiler's syntax tree and generate output code.

3.5 Extending the compiler back-end

It is possible that at some point users will want to customise the compiler so that it generates output code for a different target. The proposed software will facilitate this possibility by using a sensible and extensible framework for generating output code.

After the software has processed its syntax tree into a form which contains only valid atomic nodes, the software will pass the syntax tree through a back-end. A back-end may be any Python object which has a `generatecode()` method taking as its only argument the root node of the current syntax tree, and returning the output code.

The software will provide helper functionality to assist in writing back-ends. This will consist of a simple mechanism for writing tree traversal routines. This will probably be provided as part of the tree processing module described in Section 3.3.3.

4 Plan for completion

4.1 Current progress

This section summarises the work that I have already done towards the completion of the project.

4.1.1 Investigation of technologies

I have already spent a lot of time investigating the available technologies relating to this project. The results of this investigation are outlined in Section 2. This information will help inform me as to the tools available when completing this project and will help me to ensure that this project does not spend excessive time working on solving problems which are already solved by existing projects.

4.1.2 Language definition

Sections 3.2 and 3.3 show examples of source code for the compiler proposed in this project. As part of the work which I have undertaken already, I have drafted a formal definition of the language. Having completed this necessary step will enable me to proceed to writing and testing the parser for the system.

4.1.3 Tree manipulation library

Section 3.3.3 describes a library which provides facilities for pattern-matching and template substitution for tree structures. I have already specified, implemented and tested this library.

4.2 Milestones

The progress of the project may be measured according to the following milestones. For each milestone, the resources required for completion of that milestone and the expected duration of work on that milestone have been noted.

Milestone	Resources	Duration
Writing and testing scanner and parser.	Third-party parser generator software described in Section 2.3.	2 weeks
Preparing and presenting progress seminar.	Work to date; projector.	2 weeks
Writing and testing of main processor module.	None.	2 weeks
Writing and testing of code generator.	None.	2 weeks
Writing and testing of example modules.	None.	2 weeks
General code tidying.	None.	1 week
Write and submit abstract.	Work to date.	1 week
Prepare and present poster.	Work to date; poster printing facilities.	1 week
Final thesis.	Printing facilities.	entire year (expect about 3 weeks to compile and proof-read)

4.3 Timeline for milestone completion

The table below lists each remaining teaching week of the project and shows project milestones corresponding to these dates.

Semester/Week	Monday's Date	Milestone
S1/5	30 March	Proposal due.
S1/6	6 April	
S1/7	20 April	Complete writing and testing of scanner and parser.
S1/8	27 April	
S1/9	4 May	Complete progress seminar presentation—present at Brisbane Python User Group on 6 May.
S1/10	11 May	
S1/11	18 May	Complete writing and testing of main processor module. Present progress seminar.
S1/12	25 May	

Semester/Week	Monday's Date	Milestone
S1/13	1 June	Complete writing and testing of code generator.
S2/1	27 July	
S2/2	3 August	Complete writing and testing of example modules.
S2/3	10 August	All programming, testing and general code tidying complete.
S2/4	17 August	Begin compiling thesis write-up.
S2/5	24 August	
S2/6	31 August	
S2/7	7 September	Complete final thesis abstract.
S2/8	14 September	Complete poster.
S2/9	21 September	
S2/10	5 October	Poster and abstract due.
S2/11	12 October	
S2/12	19 October	
S2/13	26 October	Thesis due.

4.4 Risk assessment

Based on the information available at the time of writing, there are no significant risks to the completion of this project. This is justified based on the following points.

- The author has a clear idea of what the aims of the project are and what steps will be required to complete it.
- Based on the information presently available, completing the project in the required time-frame seems realistic.
- The project does not have any major dependencies on external parties or factors.
- The timeline presented in Section 4.3 allows several more weeks than are expected to be required. The timeline also assumes that all work on the project will be done during teaching weeks.

Based on these assertions it seems reasonable to concede that no major risks to project completion can be identified at the time of writing.

References

[1] A. Kennedy, "Dimension types," in *In 5th European Symp. on Programming, LNCS 788*, pp. 348–362, Springer-Verlag, 1994.

[2] A. J. Kennedy, "Relational parametricity and units of measure," in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 442–455, ACM, 1997.

[3] F. Chen, G. Rosu, and R. Venkatesan, "Rule-based analysis of dimensional safety," 2003.

[4] P. Guo and S. McCamant, "Annotation-less unit type inference for c," Dec. 2005.

[5] M. Wand and P. O'Keefe, "Automatic dimensional inference," in *Computational Logic - Essays in Honor of Alan Robinson*, pp. 479–483, 1991.

[6] T. A. Standish, "Extensibility in programming language design," *SIGPLAN Not.*, vol. 10, no. 7, pp. 18–21, 1975.

[7] G. V. Wilson, "Extensible programming for the 21st century," *ACM Queue*, vol. 2, no. 9, pp. 48–57, 2004.

[8] "PyPy[index]," Nov. 2008. `http://codespeak.net/pypy/dist/pypy/doc/`, accessed Mar. 2009.

[9] A. Rigo and S. Pedroni, "PyPy's approach to virtual machine construction," in *OOPSLA '06: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, (New York, NY, USA), pp. 944–953, ACM, 2006.

[10] A. Rigo, "Psyco—introduction." `http://psyco.sourceforge.net/introduction.html`, accessed Mar. 2009.

[11] A. Rigo, "Representation-based just-in-time specialization and the psyco prototype for Python," in *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, (New York, NY, USA), pp. 15–26, ACM Press, 2004.

[12] G. Ewing, "Pyrex—a language for writing Python extension modules." `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`, accessed Mar. 2009.

[13] S. Behnel and R. Bradshaw, "Cython: C-extensions for Python." `http://www.cython.org/`, accessed Mar. 2009.

[14] G. Ewing, "About Pyrex." `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/version/Doc/About.html`, accessed Mar. 2009.

[15] S. Behnel and R. Bradshaw, "Differences between Cython and Pyrex (Cython v0.11 documentation)." `http://docs.cython.org/docs/pyrex_differences.html`, accessed Mar. 2009.

[16] A. Kervinen, "Cinpy—C in Python," Sept. 2007. `http://www.cs.tut.fi/~ask/cinpy/`, accessed Mar. 2009.

[17] "Weave (SciPy.org)," May 2008. `http://scipy.org/Weave`, accessed Mar. 2009.

[18] "PyInline: Mix other languages directly inline with your Python," Nov. 2004. `http://pyinline.sourceforge.net/`, accessed Mar. 2009.

[19] D. K. Taft, "Google to speed up Python, add Java to app engine," *eWeek Online*, Mar. 2009. http://www.eweek.com/-c/a/Application-Development/Google-to-Speed-Up-Python-Add-Ja-va-to-App-Engine-591455/, accessed Mar. 2009.

[20] "ProjectPlan—unladen-swallow," Mar. 2009. http://code.google-.com/p/unladen-swallow/wiki/ProjectPlan, accessed Mar. 2009.

[21] D. Mertz, *Text Processing in Python.* Addison-Wesley Professional, June 2003.

[22] "mxTextTools—fast text processing for Python." http://www.egen-ix.com/products/python/mxBase/mxTextTools/, accessed Mar. 2009.

[23] eGenix.com GmbH, "mxTextTools user manual and reference guide (version 3.1)," 2008. Available for download as pdf from http://www.egenix.com/products/python/mxBase/mxTextTools.

[24] M. C. Fletcher, "SimpleParse 2.1." http://simpleparse.source-forge.net/, accessed Mar. 2009.

[25] D. Beazley, "PLY (Python lex-yacc)," Mar. 2009. http://www.da-beaz.com/ply/, accessed Mar. 2009.

[26] T. Parr, "ANTLR parser generator." http://www.antlr.org/, accessed Mar. 2009.

[27] T. Parr, "Why use ANTLR?." http://www.antlr.org/why.html, accessed Mar. 2009.

[28] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Raleigh: The Pragmatic Bookshelf, 2007.

[29] J. Aycock, "SPARK: Scanning, parsing and rewriting kit," June 2007. http://pages.cpsc.ucalgary.ca/~aycock/spark/, accessed Mar. 2009.

[30] J. Aycock, "The design and implementation of SPARK, a toolkit for implementing domain-specific languages," *Journal of Computing and Information Technology*, vol. 10, no. 1, pp. 55–66, 2002.

[31] J. Aycock and R. N. Horspool, "Practical Earley parsing," *The Computer Journal*, vol. 45, no. 6, pp. 620–630, 2002.

[32] N. Batchelder, "Python parsing tools," Mar. 2009. http://nedbatch-elder.com/text/python-parsers.html, accessed Mar. 2009.

[33] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer, "Graph transformation for specification and programming," *Science of Computer Programming*, vol. 34, no. 1, pp. 1–54, 1999.

[34] S. Wilmott, "Pattern matching in Python," Aug. 2004. http://-www.wilmott.ca/python/patternmatching.html, accessed Mar. 2009.

[35] "Gcc, the gnu compiler collection," Jan. 2009. http://gcc.gnu.org/, accessed Mar. 2009.

[36] A. Bauer, "Compilation of functional programming languages using GCC—Tail calls," Master's thesis, Institut für Informatik, Technische Universität München, 2003.