COMPILERS, THE FORGOTTEN SUBJECT?

Allan C. Milne
University of Abertay Dundee
School of Computing & Engineering Systems
Kydd Building, Bell Street, Dundee
a.milne@abertay.ac.uk
http://a510690.ces.abertay.ac.uk/

Eilidh V. McAdam
University of Abertay Dundee
School of Computing & Engineering Systems
Kydd Building, Bell Street, Dundee
e.mcadam@abertay.ac.uk

ABSTRACT

The teaching of compiler construction and language theory is absent from many current computing degrees, the rationale being that they are now irrelevant to modern software engineering practice. In this paper a case is made for the inclusion of at least certain aspects of compiler construction and language theory in computing degrees to support and reinforce the acquisition of software development and software engineering knowledge and skills in an object-oriented context. An outline curriculum based around the recursive-descent methodology is proposed and a component toolkit is described that supports the delivery of this curriculum. Small languages, formal methods and object-orientation consolidation are identified as evidence of the applicability of compiler teaching to the wider software engineering context.

Keywords

Compiling, recursive-descent, object orientation, compiler toolkit.

1. INTRODUCTION

Computing degrees may have different foci ranging from theoretical computer science to product-based information technology. A degree with "Computing" in its title should include at least a modicum of programming within a software engineering context although the depth of coverage may vary widely. Reflecting current professional software development practice this programming should be based around an object-oriented paradigm. While computer science focused courses continue to study the area of compilers and associated language theory, software-development focused courses now tend to omit this topic. In a survey of 12 Scottish universities identified as offering applied computing or software engineering degrees, only three offer modules which study compiler design and implementation. This omission is often justified from a mistaken perception that the study of compilers is now irrelevant to modern software engineering practice.

A case is made here for the inclusion of at least a restricted coverage of the areas of compiler construction and language theory as topics that are directly relevant to current object-oriented software engineering practice. This case is based on the wider applicability of the techniques acquired and to the synthesis of previously learned knowledge and skills. An outline curriculum is presented that encapsulates the compiler construction and language theory topics in the context of its applicability and synthesis. This curriculum is supported by a component toolkit that facilitates the development of an object-oriented recursive-descent compiler; this being both suitable for the implementation of a compiler for small languages and as an exemplar of the object-oriented software engineering of a system.

Compiler construction and language theory are meaningful subjects in their own right, however it must be appreciated that there is only a certain amount of space within the curriculum of a course and compilers may not be regarded as the most relevant topic for practising software engineers. Therefore a more compelling case to show this relevance must be presented for the inclusion of compiler construction and language theory in a software engineering/development focused course.

The compiler is the ultimate tool used in software development and an understanding of its operation will contribute to better programming practice, as long as this also contributes to learning in the wider software engineering context. This wider contribution is expanded upon here to indicate that inclusion of the teaching

of some aspects of compilers and languages does not “waste” curriculum space, rather it extends, reinforces and synthesizes software development knowledge and skills.

2. A PROPOSED CURRICULUM

A computer science focused course can justify a full coverage of compiler construction and language theory including such topics as grammar theory, different parsing techniques and code generation. Vocationally focused software engineering courses have a different imperative and often tend to either omit or provide cursory coverage of this area in order to address what are seen as more directly relevant subjects. A subset of the areas of compiler construction and language theory is presented here that are relevant to practical software development but yet provide a consistent view of the larger subject area. In particular the recursive-descent methodology is used in order to simplify the parsing process and provide a framework for the object-oriented compiler implementation. Theoretical aspects of grammars, parsing and code generation may be omitted without compromising a coherent and complete coverage of the compiling process in a manner that is practically relevant both directly in terms of constructing a working compiler and as a vehicle for exploring and synthesizing object-oriented development.

The curriculum can derive directly from the functional structure of a compiler in a logical and quite loosely coupled manner that can engage students by telling a “story” that, in the traditional manner, has a beginning, a middle, and an end. Some aspects of language specification must be presented first in order to give the compiler something to work with resulting in an outline curriculum schema of languages, scanning, parsing and semantic synthesis.

The curriculum presented here is based on the experience of delivering a number of different compiler courses over the last 10 years in software development focused degrees. It is flexible and can be adapted to differing course requirements; for example this curriculum has been the basis of both single and double semester courses, has been delivered using a number of different implementation languages, and has been presented at pre-final year and final year levels.

The recursive-descent compiler construction methodology is used, resulting in the straightforward creation of the controlling parser component within an object-oriented design for the overall compiler. It also clearly differentiates the roles and requirements of the functional components of the compiler in a manner that is clear to the students and emphasizes object-oriented message passing.

The aims and learning outcomes of this curriculum may be expressed in terms of the language theory and compiler construction context, in terms of software engineering concepts, or as a combination of both these domains. The author has delivered this material with aims and outcomes focusing on the compilers and languages context with the software engineering aspects being treated as a “hidden curriculum”, although this wider applicability is explicitly identified during the delivery. Alternatively, at the other end of the spectrum a course can be envisaged which focuses its aims and learning outcomes at the software engineering domain with languages and compiler construction being used to exemplify the relevant aspects of formalism and object-oriented development. The approach taken to defining these aims is an expression of the emphasis placed on the corresponding domains within the delivery rather than a reflection of a different curriculum.

The proposed outline content and delivery is as follows.

1) Language specification: BNF, derivation sequence, EBNF, LL(1) specifications.
2) Scanning: role and responsibility, token class representation, the scanner class, FSM, implementing the scanner.
3) Compiler architecture: compiler phases and structure, information flows, OO architecture, other compiler functions.
4) Parsing: role and responsibility, recursive-descent parsing structure, the parser class, transforming EBNF to recogniser methods, error handling.

5) Semantic analysis: role and responsibilities, symbols and the symbol table, class representations and architecture.
6) Artifact generation: role and responsibilities, examples of generated artifacts, generation strategies, virtual machines.

This curriculum provides a framework which a particular course may implement with differing amounts of detail depending on the general aim of the course and the time available. For example this has been delivered in a 10 week module that focused on items 1 to 5 with only a minimal consideration of artifact generation; it has also been delivered over a 20 week period with more detailed examination of items 1 to 5 and where the synthesis of artifact generation was the focus of the last quarter of delivery.

The language theory covered can focus on Backus Normal Form (BNF) and its recursive extensions in Extended BNF (EBNF) without referring to the details of grammar theory; it can state results rather than explain theory. LL(1) specifications must be covered to support the recursive-descent parsing adopted later but this can again be stated rather than explored in detail. The focus is on the practical role of BNF in defining syntax to meet the structural requirements of the domain the language is to serve.

In considering BNF as a specification language the concepts of terminal and non-terminal symbols are addressed and this leads naturally to the scanning process. At this stage the scanner is seen as a component to transform the input stream of characters of the source program into a stream of tokens representing the terminals of the program, without considering its integration into the compiler. As with all the compiler phases an object-oriented approach is taken in decomposing from the role of the components with their exposed public behaviour through to the detailed implementation of state and function using a finite state machine (FSM).

The students are then presented with the overall architecture of the compiler showing how the scanner is integrated with other classes in an object-oriented framework reflecting the functional and communication requirements of the compiler. This framework of class and object relationships then provides the basis for the detailed decomposition of each component. Figure 1 shows an outline of this object-oriented architectural framework.

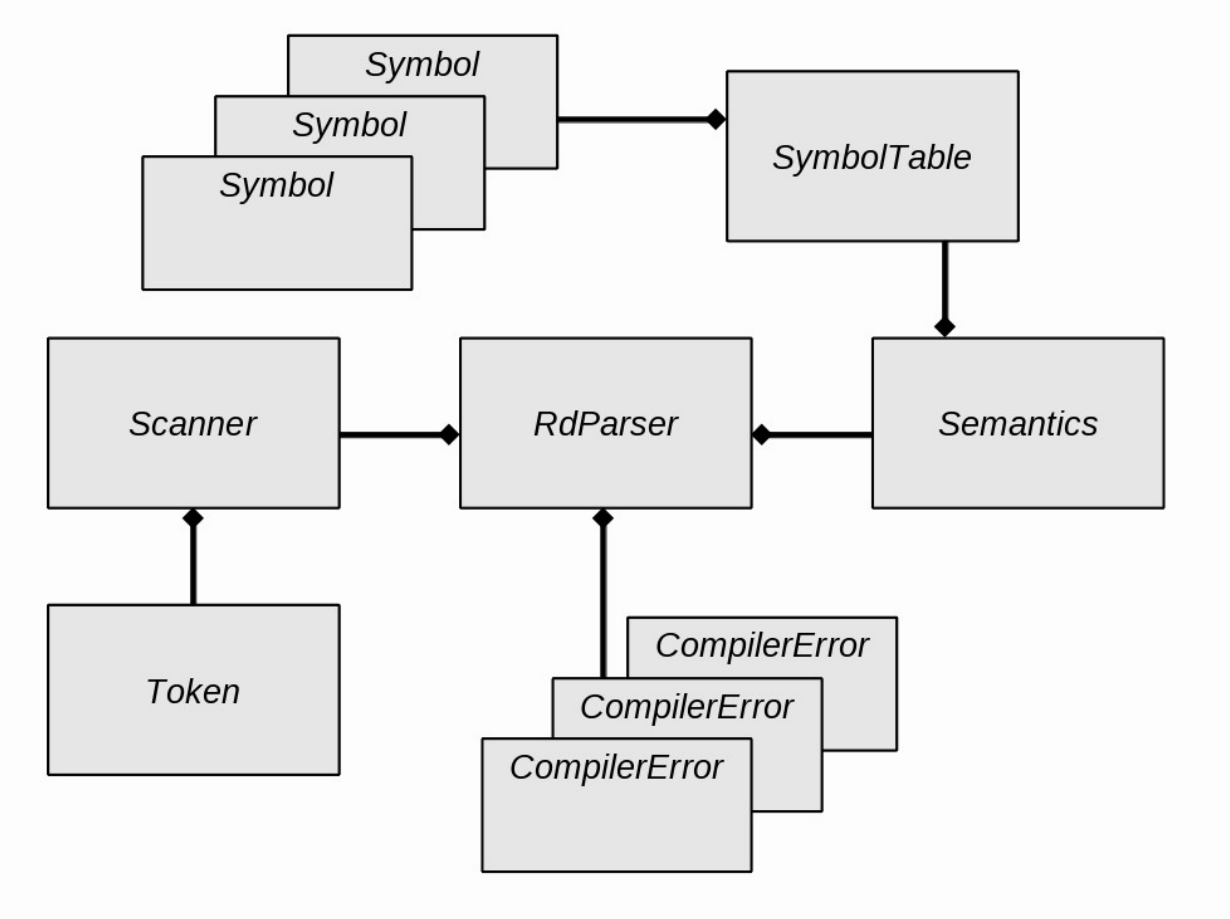

Figure 1 Object-Oriented Compiler Architecture.

With the overall picture described, the details of parsing can now be examined. Rather than covering the different top-down and bottom-up parsing techniques students are exposed to the recursive-descent methodology. This provides an example of transforming from a formal specification (the BNF) to an implementation as well as a well-defined and code-centric programming implementation. The parser is presented as the controlling component of the compiler; at this stage initiating and calling the scanning process, and as the base for the semantic analysis and artifact generation phases to follow.

The synthesis phase of the compiler, incorporating Semantic analysis and artifact generation, is focused around the role and responsibilities of the symbol table as a collection of representations of the meaning of user-defined names (identifier symbols). The teaching should first focus on these constructs then move on to the way in which they are used within the semantic analysis process. This phase is more bespoke than the preceding phases as it depends very much on the semantics of the particular language being processed. Thus teaching may cover the general architecture of integrating semantic analysis into the compiler, covering only processes typical and common to many simpler languages; alternatively, if space and time permit, the content can delve more deeply into the implementation of complex language requirements.

The requirements of artifact generation are dependent not only on the semantics of the specific source language being processed but also on the nature of the target execution platform. Traditionally this phase concerned itself with code generation for some real or virtual machine. Since many degrees no longer look at real or virtual machine architectures in any detail this would require the commitment of a large amount of learner time. Four alternative strategies have been adopted in differing guises of the delivery of this curriculum to mitigate this issue.

- Use a very simple virtual machine as the target execution platform; the one used had only 16 instructions. This minimizes the learning curve but severely limits execution mechanics.
- Select an application domain in which the target artifact to be generated is not machine-level instruction code. For example HTML has been used as a target artifact as well as other data structures for target applications with which the students are familiar.
- Generate a string containing a program in a language already known to the students and then dynamically compile this, delegating the machine-level generation to the underlying compiler. This has been done using C# for the target artifact string and then invoking dynamically the C# compiler against this string to generate an executable assembly.
- Cover this aspect of the compiler only minimally; provide an overall mechanism for connecting the artifact generation functionality into the parser but do not delve into detailed generation.

3. THE *ARDKIT* TOOLKIT LIBRARY

In presenting the curriculum outlined in the previous section in a manner that reinforces object-oriented concepts the class relationships, states and behaviour must be defined for the functional and communication components of the compiler. *Ardkit* (A Recursive-Descent toolKIT) is a library of interfaces and classes that expose the role and responsibilities of these compiler components through interface implementation, class inheritance and object composition (Milne, 2010).

Compiler generator tools have been used for many years but these are not always practical in an educational context (Demaille, 2008) as they do not expose the architecture or inner workings of a compiler and are focused on producing an output rather than exploring the underlying architecture and processes. The *Ardkit* toolkit, on the other hand, presents a library of interfaces and classes; the interfaces defining the required relationships and behaviour of compiler components; the classes providing concrete implementations of these interfaces. The toolkit classes are then used as base classes for the construction of a compiler for a specific language.

The *Ardkit* library is implemented in the C# language and exposed as a Microsoft .Net .dll library file that can be used with any language supporting the .Net framework. The toolkit is supported by full online reference and user documentation and can be viewed/downloaded at http://a510690.ces.abertay.ac.uk/Ardkit.

The presentation of the toolkit as a fully packaged and documented product is important to facilitate enquiry-led learning. Experience by the authors of other tools and resources has often been compromised by the need to present students with all the details explicitly as the operation and supporting documentation has been obscure or non-existent. The *Ardkit* library can be investigated by students with only guidance from the lecturer as to which components to review at any time.

The use of this library allows the teacher and student to focus both on the overall architecture of the compiler an, at the other end of the spectrum, the details of the implementation specific to the target language. The underlying implementation details that deal with more mundane aspects such as character input and RD primitives are hidden from the developer; of course from an educational viewpoint it may be that the implementation of the toolkit itself is worthy of inspection and the source code can be made available on request.

By studying the interfaces, classes and relationships of *Ardkit*, students will gain an understanding of the overall architecture of a recursive-descent compiler in terms of the role, responsibilities, relationships and behaviour of its functional and communication components. Students can then move to implementing their own compiler for some language by inheriting from the base *Ardkit* classes or using them directly. This allows the students to focus on those aspects of recursive-descent compiling that are specific to their language and to avoid time spent on implementing common or more basic functionality. For example, in creating a pure parser the student need only implement recogniser methods derived directly from the language's BNF specification; these are encapsulated in a class inheriting from the *Ardkit* parsing class, the latter providing all the recursive-descent primitive and client interface methods. Figure 2 outlines the only classes requiring to be implemented for many small languages.

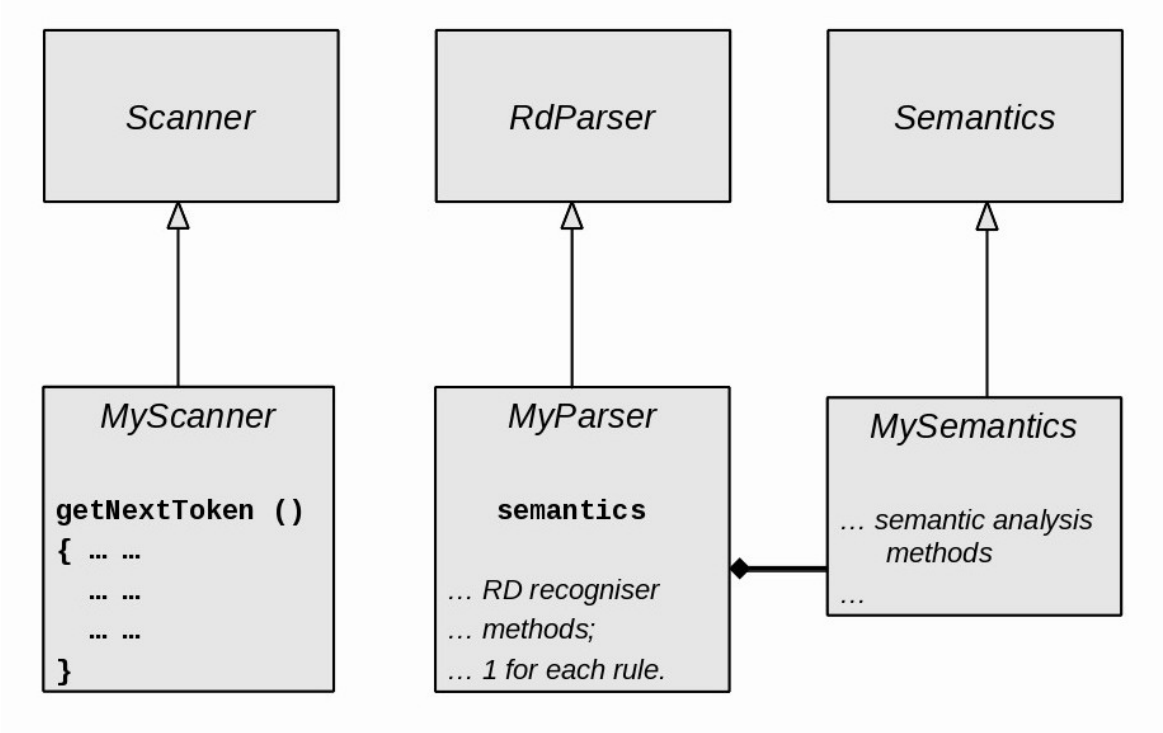

Figure 2 Implementing a Compiler using the *Ardkit* Library.

The Ardkit toolkit has been used successfully to support the teaching of the curriculum described above both in terms of examining the compiler architecture and in developing recursive-descent compilers for a variety of languages. Student feedback has been positive in terms of its efficacy in presenting and engaging students with this material and also, perhaps just as importantly, in exemplifying object-oriented design and implementation.

The toolkit is first introduced to students to illustrate the construction of a scanner for a simple language. This introduces the toolkit reference resources and the general look and feel within this simpler context of lexical analysis; only the *token* and *Scanner* classes need be considered initially. Students are therefore made comfortable with the idea of inheriting from a pre-written component based entirely its interface and reference documentation. This enables students to create scanners very quickly within a single practical session and then embed this scanner in an application to provide some basic metrics on an input program written in the original source language.

Further components of the toolkit supporting the parsing, error representation, semantic analysis and symbol manipulation requirements of the compiler are then introduced in a phased manner as each component of a compiler is presented. The toolkit classes supporting these activities are all presented in a consistent manner and students construct their compilers step by step, inheriting from the appropriate *Ardkit* component at each step.

As mentioned previously student feedback on the use of the toolkit has been positive and all students were able to construct and submit as coursework a functioning compiler for a small language. The use and feature-set of the toolkit is regularly reviewed and updated in the light of experience; currently version 2.3 has been released in March 2011. The support for synthesis and artefact generation is currently limited and is to be the next area to be reviewed. It has also become apparent that further support for language specification and processing would be useful as students often have problems in creating consistent and correct syntax specifications; this area is to be addressed by revising, extending and publishing a formal EBNF specification language and associated processing tools.

4. RELEVANCE AND APPLICATION TO SOFTWARE ENGINEERING

Here we develop the case for the inclusion of a course in compiler construction through identifying its relevance and application to the wider software engineering context. This is expressed largely in general terms; the use of the *Ardkit* library will not be specifically addressed but in all cases its use will enhance the student experience both from a practical and a software engineering viewpoint.

4.1 Small Languages

As a methodology in its own right recursive-descent compiling provides an efficient implementation technique for the construction of compilers for small languages that may be useful in some application domain. By a “small language” we mean a language that expresses the structure and semantics of some application specific entity rather than a programming language in its traditional sense.

An example of a small language is a scripting language to define the generation of a series of random numeric values of some defined type from a specified range. A script in this language might be as follows.

```
generate 10 real values from -1.99 to 1.99
repeat 25 times
   generate 2 integer values from (1,3,5,7,9)
   generate 5 real values from 0 to 99.99
   generate 1 integer value from (-999999)
endrepeat
```

This language can be defined using BNF for the syntax specification and then implementing this directly using recursive-descent parsing. Thus an understanding of basic language theory and the recursive-descent compiler construction methodology provides students with a powerful tool in their software engineering toolbox that is directly applicable to real-world application domains (Debray, 2002).

Some other practical areas of direct applicability are the specification and implementation of the structure of message protocols; scripts defining application configuration; user interface interactions. Many other examples might be enumerated but these will hopefully provide a flavour of some different areas that can be addressed by language theory and compiler construction.

4.2 Formal Methods

While formal methods are a useful tool for computing graduates, especially in the development of critical systems (Holloway, 1997), one problem in presenting them to students is illustrating the direct relevance of the formalism in a real-world practical context. The compiler and its associated language can form this practical context in which to introduce at least two formal methods; Backus Normal Form (BNF) and finite state machines (FSM).

Teaching language theory must introduce the student to BNF as a tool for defining the syntax of a language. This can be extended to a deeper discussion of grammars and grammar theory if desired although this is not necessary to support the subsequent compiler construction topic. This is also a rich base from which to discuss the role of formal and informal methods (e.g. between syntax and semantic specification) and the extension of a formalism (e.g. from grammar to BNF to extended BNF). If, as suggested, the recursive-descent compiler construction methodology is adopted then the student will also experience the direct transformation of a formal specification (the BNF) into a concrete implementation (the recogniser methods).

The use of finite state machines (or automata) is a well accepted mechanism for implementing the scanner component of a compiler. Students can be introduced to the FSM formalism through FSM diagrams and walk-through their operation in accepting input tokens built from an input stream of characters. This also links in with the further formalism of regular expressions that may or may not be examined in detail as desired. As with BNF the FSM formalism is presented within a real context, that of identifying input tokens, and this can therefore be directly experienced first through a walk-through of the process and then through creating a concrete implementation of the FSM. If deemed appropriate in the course context further details can be presented on alternative implementation patterns such as delta tables or state design patterns.

These two formalisms are exposed to the students in a practical and directly relevant manner. Introducing these two techniques into the compiling context also provides students with formalisms and patterns that are more widely applicable in the context of other application domains and problem spaces. Providing students with practical and useful examples of one formalism may indeed encourage them to investigate other formalisms in other areas of software engineering.

4.3 Synthesis of Software Engineering Knowledge and Skills

Using the recursive-descent compiling methodology and an object-oriented approach to the compiler architecture provides a framework for exemplifying and synthesising a number of software engineering concepts in a practical manner (Demaille, 2005). This introduces the students to a large-scale, component-based software application that includes a variety of different design patterns and object relationships. The *Ardkit* framework supports and facilitates this approach.

The compiler can be examined and/or developed from architectural structure to detailed code implementation in a manner that reflects software engineering development practice. This allows a very top-down approach to understanding the overall object organisation before requiring to “muddy the waters” with all the detailed program code implementation. The functional decomposition of a compiler into scanning, parsing and semantic synthesis maps well onto an object-oriented model with additional objects carrying the information flows between these functional components. This is simple enough to be wholly viewed at the top level by the student who can then also reflect on the role and responsibilities of each part of the model. This

consideration of role and responsibility without considering implementation provides a sound case study example that students can carry across to other software development projects.

When drilling down from the design to the implementation the architectural structure is also not overwhelmed by the necessity to cover sophisticated programming features as the compiler can be built from basic coding knowledge. Students are always impressed that such a sophisticated tool as a compiler can be built from simple coding features. We can then reflect on the power being expressed by the object relationships and communication rather than through the sophisticated coding. This really is a case-study of object-oriented design and patterns in action.

Thus the construction of a recursive-descent compiler provides a large-scale yet accessible object-oriented design and implementation case study distinguishing clearly between responsibilities and implementation. The design illustrates the use of inheritance and composition relationships; more detailed design introduces interfaces and abstract classes; implementation creates the concrete classes. All of this can be clearly positioned within the context of the overall compiler functionality. Therefore constructing a compiler not only creates a useful tool but also exemplifies many object-oriented design and implementation details that are directly applicable to other software development projects.

5. SUMMARY

This paper identifies a number of positive reasons for the inclusion of a course in languages and compilers in a practical computing degree. Such a course is promoted to support and reinforce the acquisition of software development and software engineering knowledge and skills in an object-oriented context.

The outline curriculum proposed illustrates that a consistent and coherent course of study can be delivered to a software engineering rather than theoretical agenda. Employing the recursive-descent methodology facilitates an object-oriented compiler architecture that enables the practical exploration of compiler construction in a top-down manner from outline design to detailed implementation. This curriculum is supported by the *Ardkit* toolkit to encourage students to focus on the larger picture and the specifics of their target language without having to worry about underlying mechanisms.

Example areas of applicability within the object-oriented software engineering context have been described where the curriculum is seen to be directly relevant to more mainstream development practices. These areas of small languages, formal methods and object-oriented case study are presented to provide a rationale for the inclusion of a compiler course in the wider curriculum.

In conclusion, it is hoped that course designers for degrees focusing on software engineering/development will consider incorporating material on compiler construction and language theory into their courses as being a relevant, positive and enabling contribution to the development of the students' software engineering knowledge and skills.

6. REFERENCES

Debray, S. (2002). Making compiler design relevant for students who will (most likely) never design a compiler, In *Proceedings of the 33rd SIGCSE technical symposium on computer science education*, ACM, New York, USA.

Demaille, A., Levillain, R. and Benoit, P. (2008) A set of tools to teach compiler construction, In *Proceedings of the 13th annual conference on Innovation and technology in computer science education,* ACM, New York, USA.

Demaille, A. (2005) Making compiler construction projects relevant to core curriculums, In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ACM, New York, USA.

Holloway, C. M. (1997). Why engineers should consider formal methods, In *Proceedings of the 16th AIAA/IEEE Digital Avionics Systems Conference.*

Mallozzi, J. S. (2005). Thoughts on and tools for teaching compiler design. *Journal of Computing Sciences in Colleges*. 21(2), pp. 177-184.

Milne, A. (2010). Ardkit Recursive-Descent Compiler Toolkit (version 2). http://a510690.cct.abertay.ac.uk/Ardkit/index.html (date accessed 23/02/2010).