An ultra-lightweight Java interpreter for bridging CS1

Stocks, Philip

Published in:
Proceedings of the 3rd Annual International Conference on Computer Science Education: Innovation and Technology CSEIT 2012

An Ultra-lightweight Java Interpreter for Bridging CS1

P. A. Stocks

School of Information Technology
Bond University
Gold Coast, QLD 4229, Australia
Email: pstocks@staff.bond.edu.au

To appear CSEIT 2012

Abstract

This paper presents an ultra-lightweight Java interpreter for use in teaching CS1 courses. The interpreter is targeted specifically at complete beginner programmers and addresses aspects particularly relevant or troublesome to novices, such as expressions, method calls, method calls as sub-expressions, and recursion. The interpreter works on a subset of Java and is intended as a bridge to a more complete environment. Experiences using the interpreter in a semester of CS1 are favourable, and an analysis of its deployment is presented.

Keywords: Java Interpreter, CS1, Introductory Programming, Computer Science Education

1 Introduction

This paper presents JULI, a Java Ultra-Lightweight Interpreter for use in introductory programming courses. The intention is to provide a more pedagogically useful learning environment for absolute beginner programmers to be used as a stepping stone to a more comprehensive programming environment, such as BlueJ. Dissatisfaction with current Java environments for teaching beginners motivates this work.

The choice of programming language and environment for teaching introductory programming is a topic of ongoing discussion and disagreement in the Computer Science community (e.g., (Mannila & de Raadt 2006)). There is pressure from both the students and the curriculum developers, in terms of necessary pre-requisites, to teach using commercial programming languages such as Java from the outset. Nevertheless, there seems general agreement in the acadaemic community that such languages are poor vehicles for pedagogy (e.g., (Allen et al. 2002, Bloch 2008, McIver & Conway 1996, Miller & Ranum 2005, Paul 2006, Radenski 2006)).

Identified requirements for an introductory teaching language are an easy syntax, a high level of abstraction, a small and consistent set of concepts and features, and a transition path to a commercial language (Kölling 1999, McIver & Conway 1996, Muziol 2008). Java fails spectacularly against all these criteria except the last. Java is a large and sophisticated language with a difficult and often counter-intuitive syntax. In particular, there is a lot of incomprehensible scaffolding required to write even the simplest working program in Java, which is a confusing barrier for new students.

Nevertheless the pressure to teach introductory programming in a language such as Java is real, and Java is currently used to teach introductory programming at Bond University, amongst many others. The choice of language to use in CS1 is not always up to the instructor. At Bond University, the role CS1 plays as a pre-requisite to other subjects requires the students to learn Java. There is no room in the curriculum for a stepping-stone course as is common practice at larger universities. This paper is not advocating which language to use to teach CS1, rather the goal is to provide a tool, and suggest an approach, to mitigate the worst problems faced by students learning CS1 using Java.

Several Java programming environments have been developed, some specifically to aid in teaching using Java. The next section examines current environments and argues the scope for a new environment and what benefits are expected from JULI. Section 3 presents the design of JULI. Section 4 presents experiences and analysis of using the trial prototype tool described in Section 3.7 in a semester of CS1. Section 5 presents directions for future work and Section 6 presents conclusions.

2 Requirements for an Introductory Programming Environment and Shortfalls of Existing Environments

The goal of this work is to provide a simple environment for learning and practicing the fundamental, and often troublesome, programming elements of expressions, methods and method calls, method calls as arguments, and recursion, without the distractions of the program scaffolding. An interpretive environment accepting snippets of Java code seems most suitable for this purpose.

Paul (2006) notes the huge benefit of a prompt feedback cycle for novice programmers, such as is provided by an interpretive environment as opposed to a compilation-based environment. Periodic moves to teach CS1 in languages such as Scheme and Python (e.g., (Bloch 2008, Agarwal & Agarwal 2005)) suggest awareness of this benefit,[1] and, indeed, Chakravarty & Keller (2004) note that perhaps the primary benefit of a move to teach CS1 using a purely functional language was the interpretive environment.

McIver & Conway (1996) identify two further qualities for an introductory programming language other than those mentioned in the previous section: that the syntax of the language/environment should be separated as much as possible from the semantics, and that better error diagnosis be available.

Current, prominent, Java interpretive environments can be improved upon when measured against these and other requirements relevant to complete beginners. BlueJ's codepad (Kölling & Rosenberg

[1] along with other benefits of using such languages

1996, Kölling 1999, Fisker & Kölling n.d.), BeanShell (*BeanShell* n.d.) Dynamic Java (*Dynamic Java* n.d.) and Dr Java (Allen et al. 2002, Stoler 2002) are considered. Other notable Java environments are the Jeliot and jGrasp visualisers (Moreno et al. 2004, Cross & Hendrix 2007), and the Ville pedagogic environment (Rajala et al. 2007). While thoroughly excellent environments, they don't, nor are intended to, work at the sub-program code snippet level, and are thus orthogonal to this discussion.

BlueJ is an excellent pedagogical environment, and its facility to instantiate and manipulate objects is extremely useful, but BlueJ is primarily a compilation environment. Recent versions have introduced the codepad interpreter, which is a useful addition that lets a learning programmer try very small code snippets before putting them into code to be compiled. Nevertheless, codepad is clearly not intended to be used alone as an initial training environment. It is a secondary piece of a larger environment. The language subset it supports is too small for significant use beyond its intended purpose. In line with its intended use in an objects first approach, it doesn't allow definition of methods, which is too limiting for it to be used in isolation. Also, and rather surprisingly given the BlueJ developers' efforts in this direction, the error reporting of codepad is very unhelpful.

BeanShell is an excellent scripting environment for quick interpretation of Java code fragments. It is not, however, intended for any pedagogical purpose, but rather for use by competent programmers. It is more complex than necessary for introductory programming purposes. The various syntactic laxities it allows, such as loose-typing, while great for experienced programmers, are very counter to requirements for an environment to learn Java and programming fundamentals. Also, its error-reporting is not useful for beginner programmers.

Dynamic Java is a flexible interpretive/scripting environment, but again is not intended for pedagogy. It has several features which are only going to be problematic for beginners. Two examples are optional variable declaration and requiring expressions to be terminated by semi-colons in order to be evaluated. This kind of syntactic training will be disastrous for complete beginners. Error reporting in Dynamic Java is, again, not helpful for beginners.

Dr Java is an excellent pedagogical compilation and interpretation environment for Java coding, following on from the success of Dr Scheme. The provision of different language levels of increasing complexity is a very attractive feature. Less attractive features of Dr Java's interpreter are that it allows duplicate variable declarations and the semi-colon laxity of allowing a statement to be entered and executed without a terminating semi-colon, unless part of a compound statement or loop body. Dr Java's interpretation of semi-colons seems to be as separators rather than the terminators they are in Java's syntax. This kind of contradictory syntactic training is harmful to complete beginners. These seem like very minor flaws, but, when added to Dr Java's error messages being not fully helpful for beginners and the fairly daunting complexity of the environment for beginners, are enough to suggest there is a place for a simpler interpretive environment only for novices and at an even lower level than Dr Java's language level 1.

Experience teaching novice programmers in Java shows that they have serious trouble grasping the most elementary aspects of the language and programming. They struggle with expressions and types and the overburdening syntax. They struggle with what should be the clear distinction between a statement and an expression, and consequently with method calls, method calls in arguments, and the difference between a `void` method and one returning a value. Finally, and consistent with the findings of Lahtinen et al. (2005), they find recursion extremely difficult.

The goal of JULI is to target these problem areas, while addressing the requirements for an introductory environment and shortfalls of existing environments outlined above. The JULI environment is at a lower level than Dr Java's language level 1 with a much simpler and more rigorous interface, designed for absolute beginner programmers only.

A key JULI feature is that it requires specific directives from the user indicating the semantic role of code before it is interpreted, thus supporting separation of syntax from semantics and enforcing learning of grammatical concepts. This reinforces conceptual aspects of programming as opposed to linguistic aspects (Paul 2006). No laxity in syntax is allowed. JULI is not intended to be used for a complete CS1 subject, but only in the first stages of the course to provide a stronger foundation in the basic elements of programming. With this foundation, the students can move confidently to a more comprehensive environment for the remainder of the course.

These two aspects of the design facilitate more precise and useful error reporting, targeted at novice programmers, since the messages can be tailored for beginners, and also since the user has declared their purpose as well as their code.

3 JULI Design

JULI is an interpreted Java programming environment designed for learning introductory programming. It is by no means a full-scale development environment, and only supports a subset of the Java language. Its purpose is to provide an environment for practising with Java expressions and simple statements in isolation from a large program. This section discusses aspects of JULI that are not features of other Java interpreters and the prototype implementation.

3.1 Language Subset

JULI understands Java at the sub class level. Arbitrary expressions and statements are understood, as well as declarations of both variables and methods. Classes cannot be defined, and access modifiers such as **public** are not understood. The keyword **static** makes no sense in this context. To JULI, everything is implicitly static. Importing libraries is also not understood, though, technically, library classes can be used through their full path class name.

Essentially, JULI is providing access to the non-object-oriented parts of Java. Classes, notably class `String`, can be used as types, instance values can be created, and methods can be called, but the definition of classes and the significance of instance variables is hidden or deferred.

JULI is intended to be used for only part of an introductory subject, to give students a grounding in the basic elements of the language and in programming fundamentals, before moving to a more complete environment. It is clearly not intended for an objects first approach. Contributing to the debate on objects first and its merits (e.g., (Bennedsen & Schulte 2007, Hu 2004)) is not a topic of this paper.

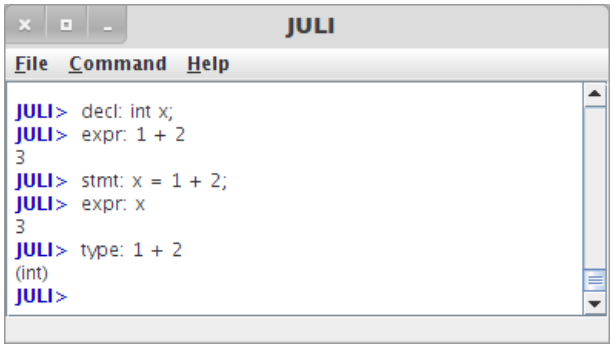

Figure 1: Basic JULI directives

There are people who do not teach objects first, and this work is aimed to support them.

Further restrictions on the language to disallow counter-intuitive aspects like assignments as expressions, not-worth-their-trouble elements like `switch` and `x++`, and the less used primitive types are possible, but currently not in place. Finally, as discussed in Section 3.4 below, standard input and output is disabled in JULI.

3.2 Interaction Through Directives

JULI is an interactive system. At the JULI prompt, the user enters directives using this simple syntax:

<directive> : [<argument>]

The directives for processing Java code are `decl`, `expr`, `stmt`, and `type`. The argument to these directives is Java code. These directives behave as one would expect. Directive `decl` expects the argument to be a variable or method declaration and enacts it. Directive `expr` expects the argument to be an expression and evaluates it. Directive `stmt` expects the argument to be a statement and executes it. Directive `type` expects the argument to be an expression and determines its type. Figure 1 shows a screenshot of a simple session showing examples of these directives.

The idea behind these directives is to force the user to understand the grammatical role and purpose of the code snippet they are entering. It is an immensely simple and easy-to-use system that reinforces the very fundamental difference between declarations, expressions, and statements. It effectively makes the distinction between a method declaration and a method call, and also clears up confusion about the difference between a call to a void method, which is a statement, and a call to a method returning a value, which is an expression.

Other directives exist for interactions with JULI rather than for processing Java code, such as for getting help, showing current declarations and statements, loading and saving scripts, and clearing declarations and statements. Each directive may be expressed in full or by provided abbreviations. For example, `expression:`, `expr:`, and `e:` are equivalent. A full list of directives is shown in Appendix A.

3.3 Sessions and Scripting

Typical scripting environments add any declaration to the currently available declarations and evaluate expressions or execute statements in the context of the current state, i.e., the values of all available variables given all the statements so far. JULI supports this behaviour, but also allows the user to declare

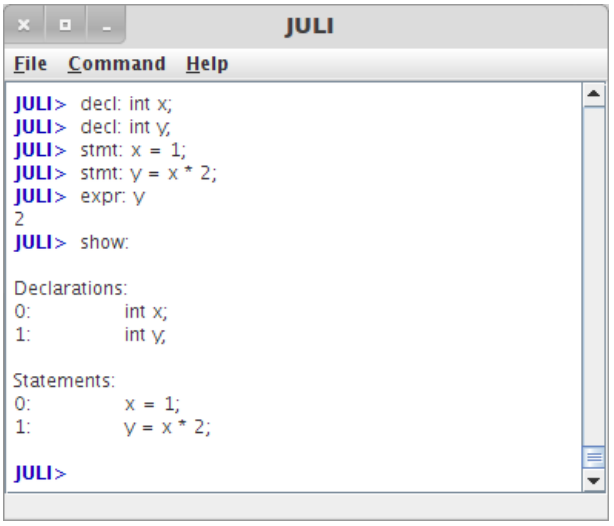

Figure 2: The `show:` directive

that declarations or statements or both are not to be remembered. When declaration recording is off, each declaration becomes the only extant declaration. When statement recording is off, each statement becomes the only statement executed and context of previous statements is lost. Since the declaration and statement directives are separate, all four possible combinations of recording are available in JULI, and there are JULI directives to set what is to be recorded.

The collection of declarations and statements currently available can be retrieved using the `show:` directive. An example is shown in Figure 2. The declarations and statements are numbered for reference in JULI directives that remove them from their respective lists.

JULI maintains a command history that can be navigated.

Finally, JULI allows the current collection of declarations and statements to be saved in a text file as a script. Similarly, scripts can be loaded from file and become the current collection of declarations and statements. The `load:` and `save:` directives, respectively, are used for this and can either be given a file path or, by default, open a file browser dialog. If the syntax of the `show:` format is used, a JULI script can be created using any text editor.

3.4 Interaction vs STDIO

McIver & Conway (1996) suggest an introductory programming environment needs to be careful with IO. The trial version of JULI does not include standard input and output. This is for conceptual reasons, and also reasons of simplicity. Looking at Java from the perspective of a program that is to be compiled, standard IO is an obvious requirement. But, from the perspective of an interpreter of Java expressions and statements, it is no longer an obvious inclusion. The rationale for not including standard IO in JULI is outlined below.

First, to be consistent with Java's operational model, standard IO should be through a console window, and this should be separate from an interpreter window.[2] That significant extra complexity needs significant justification in terms of value. JULI is intended to be used only in the beginning of a course to

[2] as is the case with BlueJ's codepad, though not Dr Java

gain familiarity with basic Java elements like expressions and methods. It is definitely not intended for writing *programs* as opposed to methods. Construction of a program as a complex state-changing entity is deferred until later in the course at a point where a different and more complete environment supporting IO can be used. Stylistically, arbitrary use of `println` statements is discouraged and instead students are encouraged to view a method as an input to output transformer through its parameters and return value. Rather than a confused jumble of `println`s across several methods, students are encouraged to construct strings as return values, and develop habits of structuring their methods calls, and future programs, accordingly. Design goals of JULI are to reinforce the concepts of expressions, method calls as expressions, and methods as type converters, similar to the experiences reported by Chakravarty & Keller (2004).

The strong desire for IO in a standard Java program is to make the system interactive, but an interpreter is interactive by nature. The default behaviour is a response, or output, and the user is always entering instructions composed of expressions, the same as input. There is actually very little utility to be had from IO in an introductory interpretive environment. This way of thinking is somewhat of a paradigm shift for those trained in the standard imperative model, but it is conceptually intuitive and beginners without such training do not find it strange.

3.5 Error Reporting

Error reporting in JULI benefits from these aspects:

1. JULI is a command interpreter. The location of the error is restricted: It must be in what the user just entered.
2. The user has declared their purpose as well as their code.
3. Error messages targetted only for complete beginners can be used.

In the event of an error, JULI shows the erroneous code highlighted, the compiler error message, and an additional explanation of what the error might be, targetted for beginners. These messages are modelled after the ones from BlueJ but even further simplified for absolute beginners.

Figure 8 shows examples of common errors.

3.6 Error Logging

JULI maintains logs of all errors made by users. Each error is tagged by a user name, a session ID, and the system time when the error occurred. This information is hashed internally so that the individual errors can be grouped appropriately, but the user is anonymous. The logs are compiled in a central, networked repository. This enables the kinds and frequencies of errors made by the students to be measured. Further, error grouping according to session and time can be measured, and thus the effectiveness of the students' attempts to resolve the errors can be gauged.

This data will also be used to tailor error reporting and handling in future versions of JULI, and will also drive additions or extensions to JULI. Furthermore, it will serve as control data for those improvements to measure their effectiveness.

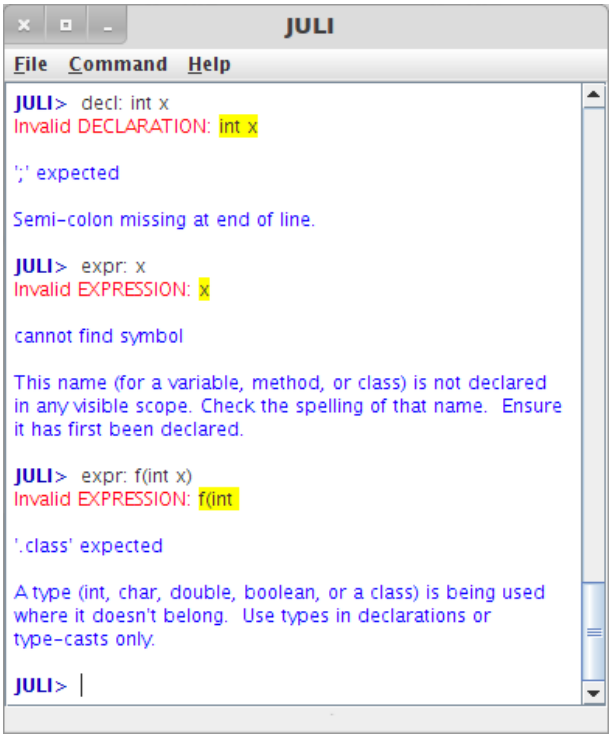

Figure 3: Error reporting

3.7 Prototype

An initial version of JULI has been implemented for trial in a CS1 class. It is implemented in Java 1.6 because it currently packages user input and ships it to the Java compiler available through the ToolProvider API. Future versions of JULI are expected also to be implemented in Java to take advantage of existing parser interfaces or to use Java's run-time type interface for creating and interacting with objects such as strings.

Figure 4 is an extended example showing a method declaration, expression evaluation, and a statement calling the method. The @ character is the prototype's temporary implementation of multi-line input. Figure 5 provides another example showing some evaluation and exploration of strings.

4 Trial Deployment and Analysis

JULI was used in a semester of introductory programming at Bond University. It was used for the first 5 lecture weeks of a 12 lecture week course to cover types and expressions, declarations and assignment, if statements, loops, method declarations and calls, recursion, and arrays. BlueJ was used for the remainder of the course, which is the environment that previous incarnations of the course have used.

There are two chiefly interesting aspects of this course arrangement. First, standard input and output is not covered until week 6 and the students don't see a complete Java program with `main` method until then. Secondly, recursion is covered very early in the course and not just as a painful afterthought at the end (or not at all) as is the case in several CS1 courses. In fact, recursion fits in naturally and seamlessly in the early stages.

The absence of standard IO was not felt amongst the students. Instead, they received a good grounding in basic expressions and statement structure.

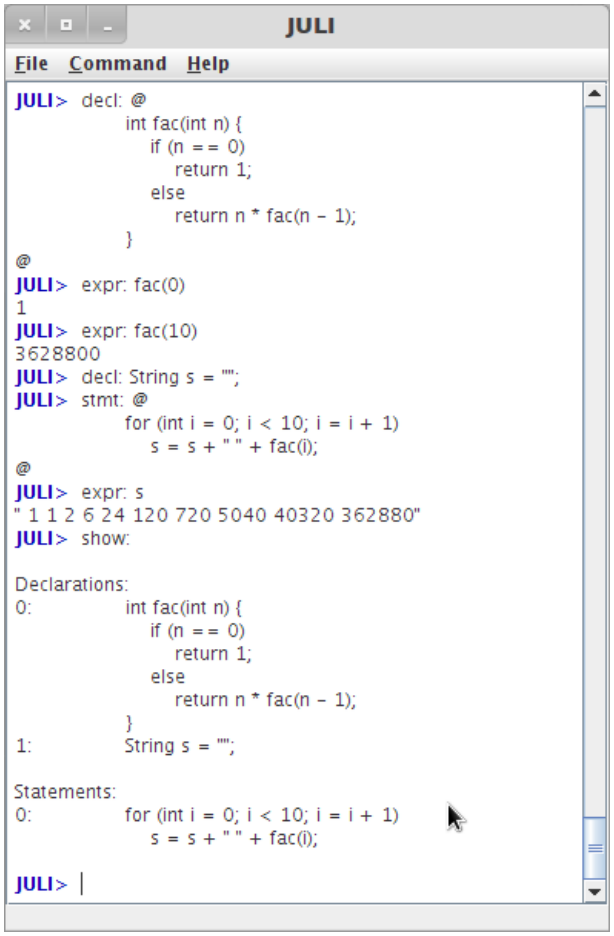

Figure 4: Method example

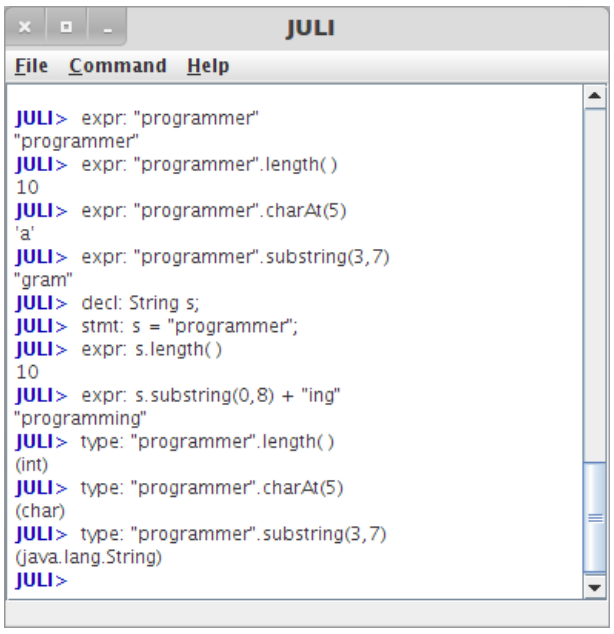

Figure 5: String exploration

Cohort	GPA	A1	A2	A3	M	P	F
May 2005	61	45	37	37	47	51	63
Sep. 2005	66	47	42	42	38	27	32
May 2006	61	74	60	56	55	46	49
Sep. 2006	65	70	92	67	70	57	69
May 2007	57	73	54	44	71	44	49
May 2008	67	68	56	55	52	59	66
May 2009	74	76	70	57	59	67	77
May 2010	68	80	74	50	57	59	74
Sep. 2010	69	81	63	71	66	64	63
Averages	65	68	61	53	57	53	60
Trial group (May 2011)	65	91	70	58	66	54	56

Figure 6: Assessment results of students using JULI. Numbers shown are percentages scored on assessment items: Three programming assignments, a Midterm exam, a Practical exam, and a Final written exam

They made the transition to full programs and BlueJ smoothly and easily, and the impression of the teaching staff is that this presentation of Java was easier to deliver and easier to receive.

In terms of the original design goals of JULI, the trial deployment was very successful. Every student, even the weak ones, fully grasped the distinction between expression and statement, that a method call returning a value is an expression, and that method calls could therefore be used in expressions and as arguments. These have been major stumbling blocks for students in previous semesters, despite special concentration and emphasis on the topics.

Furthermore, from an instructor's perspective, recursion was vastly easier to deliver with JULI and using this course structure. This is believed to be a consequence of the ability to write and call a method in isolation, and of the emphasis on method calls as expressions. The anecdotal experience of the teaching staff is that even the weak students seemed to grasp the idea, and it carried none of the arcane intimidation common in previous semesters. Nevertheless, recursion remains a difficult topic as borne out by the empirical results in the next section.

4.1 Empirical Results of Acadaemic Performance

Figure 6 summarises the acadaemic performance of students in the trial semester of JULI against previous semesters at Bond University. Each cohort is presented with the average GPA before the semester of all enrolled students, and the averages of the marks of the assessment items in the course: 3 programming assignments, a written midterm exam, a practical exam, and a written final exam. These semesters were all taught by the same lecturer and had similar assessment goals and requirements.

A standout result of Figure 6 is the trial group's performance on the first programming assignment. Different to previous semesters, this assignment was done in JULI. In terms of the problems being solved, it was identical to previous assignments. The JULI group averaged 91% for this assignment, despite having an unimpressive class GPA coming in to the subject. This solid performance continued on in assign-

Cohort	GPA	R	Cohort	GPA	R
May 2005	61	*	May 2008	67	68
Sep. 2005	66	*	May 2009	74	83
May 2006	61	*	May 2010	68	87
Sep. 2006	65	70	Sep. 2010	69	66
May 2007	57	63			
Average (May 2011)	67	72	Trial Group	65	63

Figure 7: Assessment results as percentages for recursion on practical exam (* indicates recursion not on exam).

ment 2, now doing sub-object-oriented programming with multiple (static) methods in BlueJ. The midterm exam results are also a pleasing endorsement that the basic elements of programming were better learned. The remaining results reflect full object-oriented programming and are consistent with previous years, though this group's performance on the final exam is disappointing.

One of the hopes of using JULI was that it would have a positive impact on teaching of recursion. Qualitatively, this was so, as discussed above. Quantitatively, Figure 7 shows the results for each cohort on the recursion component of the practical exam, from which it appears the trial group performed rather poorly. However, the results that contributed to this average were (50,100,0,100,100,90,0), which is a vastly more bi-modal distribution of marks than in previous semesters. In fact, the mode is 100! These results show that either students attempted the question or they did not, and when they did, they performed well. This is unlike the spread of marks in any of the other cohorts.

4.2 Empirical Results of Usage Errors

Figure 8 shows a break-down of all errors made in sessions with JULI during the trial semester. JULI only logs errors when the users are connected to the University's network, so any errors made by students on their personal machines are not represented here.

These results show the two most common mistakes are missing semi-colons (576/1477), and undeclared variables (340/1477). Together they account for 60% of all the errors, and occur an order of magnitude more often than other errors! Other Java interpretive environments (as described in Section 2) treat either or both of these as not being errors, effectively propagating 60% of mistakes to a future point where the unwitting students have been trained to believe they actually know what they are doing!

The other remarkable result from Figure 8 is that incompatible types accounted for only 4 of the 1477 errors. This is nothing short of astounding!

Error	Avg. per session	Total	Total %
Missing semi-colon	5.59	576	38.99
Undefined variable	3.30	340	23.02
Possible loss of precision	1.02	105	7.1
Illegal start of expression	0.88	91	6.16
Name already defined	0.60	62	4.2
.class expected	0.38	39	2.64
Illegal start of type	0.33	34	2.3
Can't apply symbol	0.24	25	1.69
Not a statement	0.23	23	1.56
Unclosed character literal	0.21	22	1.49
Unexpected type	0.17	18	1.22
Illegal char	0.17	18	1.22
Else without if	0.16	16	1.08
Missing return statement	0.16	16	1.08
Unclosed string literal	0.16	16	1.08
Uninitialised variable	0.12	12	0.81
Operator can't be applied	0.11	11	0.74
Illegal escape character	0.09	9	0.61
Reference to non-static	0.09	9	0.61
Repeated modifier	0.08	8	0.54
Int number too large	0.06	6	0.41
Unreachable statement	0.05	5	0.34
Incompatible types	0.04	4	0.27
Illegal line end in literal	0.03	3	0.2
Missing return type	0.03	3	0.2
Cannot dereference	0.02	2	0.14
Void not allowed here	0.01	1	0.07
Missing return value	0.01	1	0.07
Qualified name not found	0.01	1	0.07
	Total:	1477	

Figure 8: JULI usage Error classification

5 Future Work

5.1 Further Empirical Work

There is little or no empirical work contrasting Java teaching environments, nor on the general merits of an interpretive framework for teaching introductory programming. A major result of the JULI project is the acquisition of empirical data. Class sizes at Bond University are very small, so more data need to be gathered from future course offerings. Further, the error logs need to be mined for more correlations or patterns.

5.2 Further Implementation and Additional Features

Continued analysis of error logs will direct enhancements to JULI, from more constructive or refined error reports to restrictions on the input language.

In the longer term, there are two features of primary interest for adding to JULI. The first is syntax lookup, whereby a student can request the syntax diagram for a kind of statement or structure. These diagrams are used in the lecture notes and other course materials, but it would be reasonably straightforward to add them as pop-ups in JULI, and this would be more convenient for the students than having to scour course materials.

The second is a tracing/debugging component to provide a glass-box view of the system state as advocated by Robins et al. (2003). The usefulness of this concept is clear, but BlueJ already provides an excellent debugging environment, especially when combined with Jeliot. Given that JULI is only intended for initial use before transferring to an environment such as BlueJ, this feature has lower priority.

5.3 Broader Deployment

The full implementation of JULI will be freely available for other teachers of CS1 classes to use. It is distributed as a `jar` file, so it should also be easy to incorporate it into BlueJ as an extension. A more wishful deployment prospect is to see the ideas of JULI's directives incorporated into Dr Java as some kind of plug-in language level 0 beneath level 1.

6 Conclusion

The design and usage experiences of an ultra-lightweight Java interpreter for the early stages only of an introductory programming subject have been presented. The interpreter, which understands Java at the sub class level, requires the user to precede their code snippets by an indication of their grammatical role. These two ideas target complete novices and also allow the error reporting to be tailored specifically to help beginners. JULI is concerned with trying to solve a small problem effectively: that current Java pedagogical environments are still too complex or have features unsuitable for absolute beginners. Environments like BlueJ and Dr Java are thoroughly excellent, and this work is by no means a criticism of them. Indeed, the intention is to begin with JULI and transition to such environments when the students have a firm grounding in fundamental aspects of programming and Java.

Beyond this role of a stepping stone, JULI also tries to target perceived related problem areas for beginning students of programming in Java: expressions versus statements, methods, method calls, methods as functions and type converters, method calls in expressions, and, finally, recursion. The functional programming flavour of this is deliberate. JULI provides a way to access sub class level Java in a simple and, if desired, declarative way.

Experience with JULI in a semester of CS1 was very positive, and empirical results suggest it had a positive impact on learning outcomes. JULI sits well in an introductory programming curriculum for covering basic elements before moving to a more complete environment for larger programs.

References

Agarwal, K. K. & Agarwal, A. (2005), 'Python for CS1, CS2 and beyond', *Journal of Computing Science in Colleges* **20**.

Allen, E., Cartwright, R. & Stoler, B. (2002), Dr Java: A lightweight pedagogic environment for Java., *in* 'Proceedings of the ACM 33rd SIGCSE Technical Symposium on Computer Science Education.'.

BeanShell (n.d.), `http://www.beanshell.org`.

Bennedsen, J. & Schulte, C. (2007), What does "objects-first" mean? An international study of teachers' perceptions of objects-first, *in* 'Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)'.

Bloch, S. (2008), 'Teach Scheme reach Java: Introducing object-oriented programming without drowning in syntax', *Journal of Computing Science in Colleges* **23**.

Chakravarty, M. M. T. & Keller, G. (2004), 'The risks and benefits of teaching purely functional programming in first year', *Journal of Functional Programming* **14**, 113–123.

Cross, J. A. & Hendrix, T. D. (2007), 'jGRASP: An integrated development environment with visualizations for teaching Java in CS1, CS2, and beyond.', *Journal of Computing in Small Colleges* **23**(2).

Dynamic Java (n.d.), `http://old.koalateam.com/djava/`.

Fisker, K. & Kölling, M. (n.d.), 'The BlueJ environment reference manual (v2.0)', `http://www.bluej.org/doc/bluej-ref-manual.pdf`.

Hu, C. (2004), 'Rethinking teaching objects-first', *Education and Information Technologies* **9**(3).

Kölling, M. (1999), The design of an object-oriented environment and language for teaching., PhD thesis, The University of Sydney.

Kölling, M. & Rosenberg, J. (1996), An object-oriented program development environment for the first programming course., *in* 'Proceedings of ACM 27th SIGCSE Technical Symposium on Computer Science Education.'.

Lahtinen, E., Ala-Mukta, K. & Järvinen, H.-M. (2005), A study of the difficulties of novice programmers, *in* 'Proceedings of the 10th International Conference on Innovation and Technology in Computer Science Education (ITiCSE'05)'.

Mannila, L. & de Raadt, M. (2006), An Objective Comparison of Languages for Teaching Introductory Programming, *in* 'Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006'.

McIver, L. & Conway, D. (1996), Seven deadly sins of introductory programming language design., *in* 'Proceedings of the 1996 International Conference on Software Engineering: Education and Practice.'.

Miller, B. N. & Ranum, D. L. (2005), Teaching an Introductory Computer Science Sequence in Python, *in* 'Midwest Instruction and Computing Symposium'.

Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004), Visualizing programs with Jeliot 3., *in* 'Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004, Gallipoli (Lecce), Italy'.

Muziol, L. (2008), 'Teaching programming : Modern approaches using tools and dynamic languages.', `http://gride.googlecode.com/files/lmuziol-teaching-summary.pdf`.

Paul, J. (2006), What first? Addressing the critical initial weeks of CS-1., *in* 'Proceedings of the 36th ASEE/IEEE Frontiers in Education Conference'.

Radenski, A. (2006), "Python First": A Lab-Based Digital Introduction to Computer Science, *in* 'Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education', ITICSE '06.

Rajala, T., Laakso, M.-J., Kaila, E. & Salakoski, T. (2007), VILLE - a language-independent program visualization tool., *in* 'Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)'.

Robins, A., Rountree, J. & Rountree, N. (2003), 'Learning and teaching programming: A review and discussion', *Computer Science Education* **13**, 137–172.

Stoler, B. (2002), A framework for building pedagogic Java programming environments., Master's thesis, Rice University.

A JULI Directives

JULI commands
(abbreviations shown in parentheses)

```
clear :

    Clear all current declarations and
    statements (see show below)
    (c)

clear-declaration :  <arg>

    Clear the given declaration number,
    or all declarations if no argument
    (cd, clear-decl)

clear-statement :  <arg>

    Clear the given statement number, or
    all statements if no argument
    (cs, clear-stmt)

declaration :  <code>

    Interpret the given code as a
    declaration
    (d, dec, decl)

expression :  <code>

    Interpret the given code as an
    expression
    (e, expr)

header :  <arg>

    Add a header comment to the code
    (head)

help :

    Show this help message
    (h)

load :  <filename>

    Load declarations and statements
    from file.  Use <filename> if
    present, otherwise open browser.

record :  <on|off>

    Set the recording status for
    declarations and statements to on
    or off.  If recording is on, each
    new declaration or statement is
    appended.  If recording is off,
    each new declaration or statement
    replaces the last.  The default
    status is off.
    (r, rec)

record-status :

    Show the record status for
    declarations and statements
    (rstat, rec-status)

record-declarations :  <on|off>

    Set the recording status for
    declarations to on or off (see
    record above).
    (rd, rec-dec, rec-decl, record-dec,
    record-decl)

record-statements :  <on|off>

    Set the recording status for
    statements to on or off (see record
    above).
    (rs, rec-stmt, record-stmt)

record-toggle :

    Toggle the record status for
    declarations and statements
    (on->off, off->on)
    (rt, rec-toggle)

run :

    Execute the current statements
    in the context of the current
    declarations

save :  <filename>

    Save the current declarations and
    statements to file.  Use <filename>
    if present, otherwise open browser.

show :

    Show the current declarations and
    statements
    (sh)

statement :  <code>

    Interpret the given code as a
    statement
    (s, st, stmt)

type :  <code>

    Show the type of the given
    expression
    (t)
```