

Li Xu
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01854, USA
978-934-1941
xu@cs.uml.edu

ABSTRACT

Language engineering – the theory and practice of compiler and language processor design, has long been recognized as a core subject in Computer Science curricula. However, due to the scope and depth of the included topics, the limited time constraint, and the lack of suitable target system, educators often find it difficult to engage students and teach language engineering courses effectively.

This paper presents a new modular approach to language engineering using XML and inexpensive robots. We teach language engineering in the context of the popular, inexpensive Scribbler robot, and use XML to externalize the compiler intermediate representation. The inexpensive robot device provides an affordable, yet realistic system platform to explore a broad set of compiler topics; the XML-based intermediate representation divides the compiler components into self-contained, independent modules; combined together, these two techniques give instructors a highly flexible and powerful framework to design course materials and teach language engineering subjects.

We have designed the Chirp-Scribbler language to program the Scribbler robots. The Chirp-Scribbler compiler consists of lexer, parser, semantic verifier, code generator, and uses XML as shared intermediate representation. Each of the components can be built as a separate module and integrated to create the complete compiler. We describe the design of Chirp-Scribbler language, the XML-based module structure of Chirp-Scribbler compiler, integration with robots and supporting tools, and our teaching practice of using them to teach language translation basics in an undergraduate programming course.

INTRODUCTION

Language engineering teaches the theory and practice of compiler and language processor design. Its importance and benefits are well documented [7, 13, 14, 19], and it has long been recognized a core subject in Computer Science curricula. All past ACM Computing Curricula (Curricula 1968, Curricula 1978, Curricula 1991) listed compiler and language implementation as main curriculum topics [4, 5, 6]. In the latest Computing Curricula 2001 (CC'2001), which strives toward broadening Computer Science education with emerging subjects, listed Language Translation Systems and Compiler Construction as a single upper-level elective course (Unit PL8 in the Programming Language area), while the Introduction to Language Translation (Unit PL3) is listed as required core subject [12].

As a well developed subject domain, compiler and language engineering have become a rich and sophisticated subject of study in its own right. The widely used compiler textbooks [7, 13] have a broad range of topics, including lexing, parsing, intermediate representation, code generation, and compiler optimization, and require two compiler classes, an introductory compiler course and an advanced second compiler class, for full coverage. However, as CC'2001 reduces time on compiler and language translation subjects, either as an one-course elective (Unit PL8), or as a topic unit (PL3), instructors must carefully choose a selective set of key concepts as topics and balance the depth and breadth of the course, so it can fit with the goals and time constraint by the mandated curriculum.

In addition to the above curriculum and time constraints, another challenge to teach language engineering is the lack of suitable system context. Modern compilers are increasingly built as an integral part of the whole system (system tool chain or run-time) [13, 18, 11]; however, in an academic setting, due to lack of resources, language engineering is typically taught without the use of real target systems. Students implement contrived

teaching languages and use simulators, rather than use real language and real system as the target; student projects are designed to understand the selected concepts, but rarely do students have the opportunity to explore these concepts in the context of complete real systems.

Leveraging the XML data format and emerging inexpensive educational robots, we have developed a modular approach to teach language engineering subjects. We have designed the Chirp-Scribbler language to program the popular Scribbler robot. The internal representation of the Chirp-Scribbler compiler is implemented using XML trees and saved in XML files. Key compiler components, such as lexer, parser, semantic verifier, and code generator, are built as separate modules processing the shared XML-based intermediate representation (IR). Using this approach, instructors can easily select and mix suitable modules to teach compiler and language translation topics with robots as the concrete target system.

The rest of the paper is organized as follows: Section 2 describes the background of Scribbler robot, the Chirp-Scribbler language, and XML-based compiler IR and processing using JDOM; Section 3 describes the XML-based Chirp-Scribbler compiler; Section 4 describes the integration of Chirp-Scribbler compiler with the Scribbler robot; Section 5 describes our experience of using Chirp-Scribbler in an undergraduate programming course; we conclude in Section 6.

BACKGROUND

Scribbler Robot



Components	Spec
Processor	8-bit Microchip PIC16C57c
Speed	20 MHz
RAM	32 Bytes
EEPROM	2K Bytes
I/O Pins	16
PBASIC Commands	42
Light Sensors	3
Obstacle Sensors	3 (2 emitters, 1 detector)
Line Sensors	2
Stall Sensor	1
DC Motors	2
LED Lights	3
Speaker	1
Serial Port	1

Figure 1: Parallax Scribbler robot and hardware specification

The emerging inexpensive educational robots have been valuable teaching tools to Computer Science educators [8]. Products such as LEGO Mindstorms RCX and NXT robots [21, 23], Parallax Scribbler robots [20], and MIT Handyboard and Handy Cricket [27] have been widely used to teach Artificial Intelligence and Robotics [25, 10, 9, 24, 22], and more recently general Computer Science courses [16, 15, 29, 17, 23].

We believe these inexpensive robot devices can play an equally valuable role in language engineering courses. Their affordable price makes it possible for large-scale adoption in an undergraduate course. In our class, we are able to give each student a Scribbler robot kit to use (under \$100 per unit). Furthermore, they provide an easy to use and realistic system context, and can be used to teach both introductory and advanced topics, in particular, topics on modern compiler backend design, which require a real target system to make it practical.

Figure 1 shows the Scribbler robot and its hardware specification. Scribbler runs on an embedded BASIC Stamp 2 (BS2) microcontroller. As a highly constrained embedded system, Scribbler has only 32 bytes RAM

<pre> const int do = 391; // note Do's frequency const int re = 494; // note Re const int mi = 523; // note Mi int tune_time; bit stalled; pin(11) speaker; void startTune() { tune_time = 200; // 200ms System.sound(speaker, tune_time, do); System.sound(speaker, tune_time, re); System.sound(speaker, tune_time, mi); } void checkState() { Scribbler.senseStall(stalled); if (stalled) { Scribbler.setLED(1, 1, 1); Scribbler.moveBackward(5, 5, 1000); Scribbler.moveLeft(5, 5, 1000); } else { Scribbler.setLED(0, 0, 0); } } void main() { startTune(); loop { Scribbler.moveForward(5, 5); checkState(); } } </pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <AST_IR> <VAR_DEF const="391"> <type>int</type> <name>do</name> </VAR_DEF> <VAR_DEF const="494"> <type>int</type> <name>re</name> </VAR_DEF> <VAR_DEF const="523"> <type>int</type> <name>mi</name> </VAR_DEF> <VAR_DEF> <type>int</type> <name>tune_time</name> </VAR_DEF> <VAR_DEF> <type>bit</type> <name>stalled</name> </VAR_DEF> <VAR_DEF> <type pin_num="11">pin</type> <name>speaker</name> </VAR_DEF> <FUNC_DEF> <name>startTune</name> <Block> <ASSIGN_VAR> </pre>
---	---

Figure 2: Chirp-Scribbler sample program and abstract syntax tree (AST) in XML

(for program variables) and 2K bytes ROM (code and data). The BS2 microcontroller has 16 programmable I/O pins which connect to Scribbler's 9 sensors, 2 DC motors, 3 LEDs, an on-board speaker and serial port for PC communication. Parallax provides a BASIC-style low-level command set (42 PBASIC commands) to program the robot [26, 28].

The Chirp-Scribbler Language

To teach language engineering with Scribbler, drawing inspiration from the Chirp language [30, 31], we have designed the Chirp-Scribbler Language for programming the Scribbler robot.

Chirp-Scribbler has syntax similar to C. It has typical imperative language features such variables, constants, functions, control statements and expressions. Targeting the Scribbler robot, it also defines data types and control constructs to target the embedded BS2 microcontroller. For example, to access bit and pin data, it has `bit` and `pin` types; all Chirp-Scribbler variables are global, as there is no stack memory on BS2. Such features illustrate idiosyncrasies of real languages for embedded systems, and demonstrate the close interaction between compiler and the target system.

An example Chirp-Scribbler program is shown in Figure 2. It consists of variable definitions and three functions (including `main` function): the robot starts by playing music notes on the speaker pin (Pin 11), then starts the motors and moves forward; it checks the stall sensor to see if it hits obstacles, and if so, it

turns on all three LEDs, backs away and turns left.

XML-based Compiler IR and JDOM

The eXtensible Markup Language (XML) is a W3C standard for general purpose textual data representation and markup [3]. Each valid XML document has a tree structure, with the root representing the top-level element and internal nodes as the subordinate elements. To build the Chirp-Scribbler compiler, we use XML as intermediate representation (IR), for example, the abstract syntax tree (AST) of the sample program is encoded in XML and shown on the right side in Figure 2.

To construct and manipulate XML-based tree IR, we use the open-source JDOM library [2]. JDOM provides a simple Java object model to represent XML document. We use the JDOM Document and Element classes to build XML document tree and access element nodes.

THE CHIRP-SCRIBBLER COMPILER

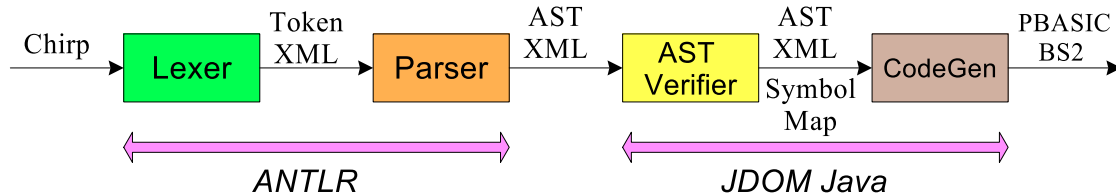


Figure 3: Chirp-Scribbler compiler modules

Compiler Structure

The structure of the Chirp-Scribbler compiler is shown in Figure 3. Following the general structure of modern compilers [13], the Chirp-Scribbler compiler consists of the following modules:

- **Lexer:** the Lexer translates Chirp-Scribbler source program into token stream. The Lexer is built using the ANTLR compiler generator tool [1]. The token patterns are specified using ANTLR grammar rules and ANTLR generates the Lexer itself. The Lexer token stream output is used to build token XML tree to represent the tokens.
- **Parser:** the Parser reads the token stream, parses it and builds the abstract syntax tree (AST) IR. We choose the ANTLR tool to construct the Parser as ANTLR generates easy to understand recursive-descent parser. We define the parser grammar rules in the ANTLR rule file; for each of the parser rule, we associate Java action code to build the XML-based AST using JDOM. The AST XML tree is saved in an external XML file and passed to the back end.
- **AST Verifier:** the AST Verifier uses JDOM to read the AST XML file and reconstruct the AST XML tree. It then performs tree walk on the variable and function elements to check semantics of the Chirp-Scribbler program. It also builds symbol map information for use by the Code Generator.
- **Code Generator:** the Code Generator also uses JDOM to build the XML-based AST and visits the variable and function definition elements to generate BS2 PBASIC code.

In the above, each module uses XML as shared compiler IR and the XML tree is saved as XML file between passes. This design allows compiler modules be constructed independently and gives the instructor maximum flexibility and control over Chirp-Scribbler. For example, in code generation, instructor can provide only the AST XML file, students will use JDOM to recreate the AST and explore code generation without the need to deal with front end issues. Instructor can also give out reference modules in binary form and let students build the required module to realize the whole compiler.

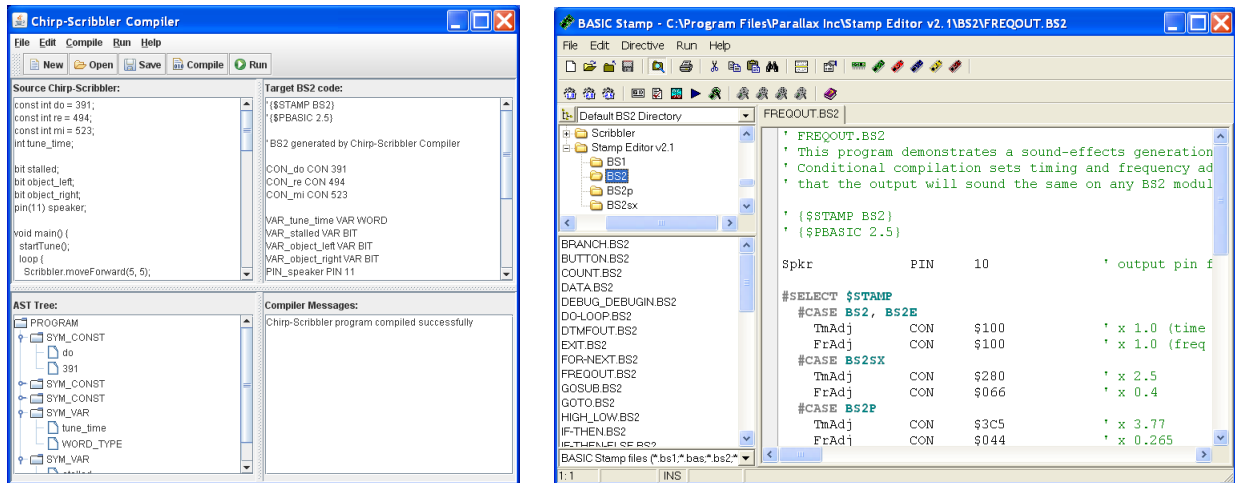


Figure 4: The Chirp-Scribbler Compiler GUI and BASIC Stamp Editor

Compiler IR Visualization using XMLTreeView

The XML-based IR captures the internal state of the compilation process, and can be used to visualize how compiler works, as well as a debugging tool. The token IR and AST IR XML files represent the lexer token stream and parser AST through XML textual format, and can be used directly as artifacts to understand compilers. To help visualize the XML-based IR, we developed the XMLTreeView utility class. It builds on the JDOM JTreeOutputter, and creates Java JTree GUI to display the JDOM Document and Element objects. To visualize any XML tree node, user can simply create an XMLTreeView instance to display the tree node in Java.

COMPILER INTEGRATION WITH ROBOT

The Chirp-Scribbler compiler integrates closely with the target robot platform. Its GUI interface models that of the Parallax BASIC Stamp Editor tool, and includes display panels for the source code editor, the generated BS2 commands, XMLTreeView of the AST and compiler messages. The generated BS2 commands are then passed as input to the Stamp Editor tool and run on the Scribbler robot. Using Scribbler as the system target demonstrates the whole-system perspective and compiler as part of the system tool chain [11]. The Chirp-Scribbler Compiler and the Basic Stamp Editor tool are shown in Figure 4.

USING CHIRP-SCRIBBLER IN A PROGRAMMING COURSE

We have been using Chirp-Scribbler to teach language engineering components in an intermediate undergraduate programming course at our institution. The class includes language translation basics as listed in Unit PL3 in CC'2001.

Our preliminary results indicate Chirp-Scribbler has been an effective teaching tool for language engineering topics. In addition, students interest levels are high, and students love to explore their code on Scribbler. We are currently evaluating our findings and will publish the results in a followup report.

CONCLUSIONS

This paper presents a modular approach to language engineering using XML and inexpensive robots. We have designed the Chirp-Scribbler language and used XML-based compiler intermediate representation to

build the Chirp-Scribbler compiler. By targeting the Scribbler robot as real system context, it provides a flexible and powerful framework to teach language engineering subjects.

REFERENCES

- [1] ANTLR Documentation. <http://www.antlr.org/doc>.
- [2] JDOM Documentation. <http://www.jdom.org/>.
- [3] W3C XML Specification. <http://www.w3.org/XML>.
- [4] ACM Curriculum Committee on Computer Science. Curriculum '68: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, pages 151–197, Mar. 1968.
- [5] ACM Curriculum Committee on Computer Science. Curriculum '78. *Communications of the ACM*, pages 147–166, Mar. 1979.
- [6] ACM Curriculum Committee on Computer Science. Computing Curricula 1991. *Communications of the ACM*, pages 68–84, June 1991.
- [7] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [8] D. Blank. Robots make computer science personal. *Communications of the ACM*, 49(12):25–27, 2006.
- [9] D. Blank, D. Kumar, L. Meeden, and H. Yanco. Pyro: A python-based versatile programming environment for teaching robotics. *Journal on Educational Resources in Computing*, 3(4):1–15, 2003.
- [10] D. Blank, L. Meeden, and D. Kumar. Python robotics: an environment for exploring robotics beyond legos. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 317–321, 2003.
- [11] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.
- [12] CC2001 Task Force. Computing Curricula 2001, Computer Science Volume. <http://www.sigcse.org/cc2001/>, Dec. 2001.
- [13] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2003.
- [14] S. Debray. Making compiler design relevant for students who will (most likely) never design a compiler. In *SIGCSE '02: Proceedings of 2002 SIGCSE technical symposium on Computer science education*, pages 341–345, 2002.
- [15] B. Fagin and L. Merkle. Measuring the effectiveness of robots in teaching computer science. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 307–311, 2003.
- [16] B. S. Fagin and L. Merkle. Quantitative analysis of the effects of robots on introductory computer science education. *Journal on Educational Resources in Computing*, 2(4):2, 2002.
- [17] M. Goldweber, C. Congdon, B. Fagin, D. Hwang, and F. Klassner. The use of robots in the undergraduate curriculum: experience reports. In *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 404–405, 2001.
- [18] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition*. Morgan Kaufmann, 2002.
- [19] T. R. Henry. Teaching compiler construction using a domain specific language. In *SIGCSE '05: Proceedings of 2005 SIGCSE technical symposium on Computer science education*, pages 7–11, 2005.
- [20] Institute for Personal Robots in Education. <http://www.roboteducation.org/>.
- [21] F. Klassner. A case study of lego mindstorms' suitability for artificial intelligence and robotics courses at the college level. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 8–12, 2002.
- [22] F. Klassner. Enhancing lisp instruction with rxcslisp and robotics. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 214–218, 2004.
- [23] F. Klassner and C. Continanza. Mindstorms without robotics: an alternative to simulations in systems courses. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 175–179, 2007.
- [24] A. N. Kumar. Three years of using robots in an artificial intelligence course: lessons learned. *Journal on Educational Resources in Computing*, 4(3):2, 2004.
- [25] D. Kumar and L. Meeden. A robot laboratory for teaching artificial intelligence. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 341–344, 1998.
- [26] A. Lindsay. *What's a Microcontroller*. Parallax, Inc.
- [27] F. Martin, B. Mikhak, and B. Silverman. Metacrickit: A designer's kit for making computational devices. *IBM Systems Journal*, 39(3 & 4), 2000.
- [28] Parallax, Inc. *BASIC Stamp Reference Manual, version 2.1*.
- [29] E. Sklar, S. Parsons, and P. Stone. Using robocup in university-level computer science education. *Journal on Educational Resources in Computing*, 4(2):4, 2004.
- [30] L. Xu and F. Martin. The chirp language specification. Technical Report TR-2005-003, Dept. of Computer Science, UMass Lowell.
- [31] L. Xu and F. G. Martin. Chirp on crickets: teaching compilers using an embedded robot controller. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 82–86, 2006.