# Chirp on Crickets: Teaching Compilers Using an Embedded Robot Controller

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

Fred G. Martin
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01854 USA
fredm@cs.uml.edu

## ABSTRACT

Traditionally, the topics of compiler construction and language processing have been taught as an elective course in Computer Science curricula. As such, students may graduate with little understanding or experience with the useful techniques embodied in modern compiler construction.

In this paper, we present the design of Chirp, a language specification and compiler implementation. As a language, Chirp is based on Java/C syntax conventions and is matched with the stack-based virtual machine that is built into the simple yet versatile Handy Cricket educational robot controller. As a compiler, the Chirp design is a series of Java components. These modules demonstrate key compiler construction techniques including lexing, parsing, intermediate representation, semantic analysis, error handling and code generation.

We have designed a 6-week teaching module to be integrated into an intermediate-level undergraduate programming class. In the module, students will incrementally build the Chirp compiler, culminating with code generation for the Cricket controller. They will test their work on both physical Cricket-based robots and a web-based Cricket simulator. The Chirp system and our pedagogical design provides a realistic and engaging environment to teach compilers in undergraduate core programming courses.

## Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]; D.3.4 [**Programming Languages**]: *Compilers*

## General Terms

Languages, Design

## Keywords

Compiler Construction, Embedded Processor, Educational Robotics, Virtual Machine, Language Processing

## 1. INTRODUCTION

Compiler construction has long been included in the computer science curricula. Due to its complexity, however, the compiler course is usually available only as an upper-level elective class. Students often opt not to take the course, thus missing the opportunity to acquire the knowledge and skills embodied in a compiler class—many of which are essential for understanding the concepts and inner workings of languages and programmable systems.

This paper presents a teaching framework to integrate the fundamentals of compiler and language processing into an intermediate-level programming course. Based on the Handy Cricket embedded robot controller, we developed a C-like language dubbed Chirp. The Chirp language and compiler system target Cricket robots and a web-based Cricket simulator and provide a simple, yet realistic and engaging framework to teach compilers.

This paper makes the following contributions:

1. We present a new approach to integrate teaching compilers into an intermediate-level undergraduate programming course, so more students will benefit from learning fundamentals in compiler design;
2. The Chirp language and its compiler are designed for pedagogy, and give students balanced and accessible coverage on broadly applicable compiler techniques, including front-end, back-end and intermediate representation in typical compiler development;
3. We present a teaching module based on the Chirp framework and the supporting tools and techniques, so the materials can be effectively taught within the context of intermediate-level core programming course.

The rest of the paper is organized as follows: Section 2 describes the motivation of our approach. Section 3 describes the Handy Cricket embedded robot controller and the Chirp language. Section 4 describes the Chirp compiler design. Section 5 highlights the tools and techniques to assist students building the compiler. We conclude in Section 6.

## 2. MOTIVATION AND RELATED WORK

Compiler Construction has been included in Computer Science curricula since the early days of computing [3, 4]. The ACM Curricula 2001 [1] lists Language Translation Systems as elective in the Programming Languages knowledge group. The widely used compiler textbooks [5, 7] suggest a primary teaching format of a dedicated compiler class to

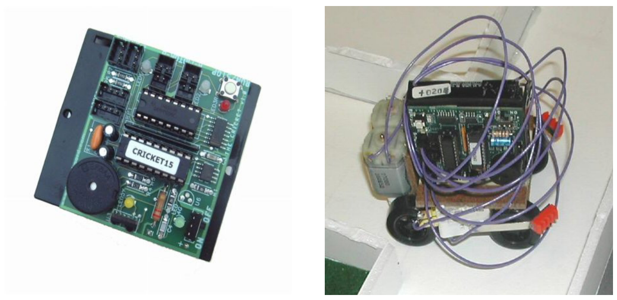


**Figure 1: The Handy Cricket Controller and Maze-Crawling "Mini-Bot."**

cover the main materials, and a second class to teach more advanced topics such as program analysis and optimization. Such a format is currently used in many institutions. As noted, this elective class approach limits the audience to a small number of students, as the majority of students opt not to take the class at all.

Many educators recognize the importance of teaching compilers in undergraduate courses, and the need to include more students in such a course. Debray [8] shows compiler techniques are highly useful with a wide variety applications and advocates a generalized view in teaching compilers. Henry [9] describes teaching compilers using domain specific languages to engage students and to demonstrate the usefulness of compilers. Nguyen *et al.* [14] use design patterns to teach recursive-descent parsing in a CS2-level OO programming course.

Although few students will likely become professional compiler writers, the principles and techniques introduced in compiler class are broadly applicable. For example, compiler front-end has its foundation in automata, grammar and parsing theory. Compiler design involves non-trivial use of common data structures and algorithms. The techniques of language processing can be used to solve many non-compiler related problems [7, 5, 8]. Good knowledge of these topics are essential for students to understand concepts and inner workings of languages and language-based systems.

To allow more students to benefit from learning compilers, an effective approach is to integrate compiler fundamentals into the core curriculum. The challenge of this approach, however, is to carefully select subjects so that students will learn the core topics without being overwhelmed by advanced topics. In our work, the Chirp framework is developed to give students balanced and accessible coverage on key compiler techniques in the context of an intermediate-level programming course.

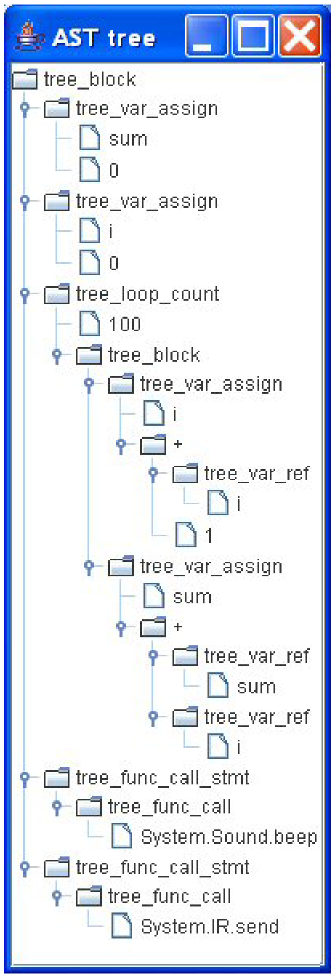
## 3. HANDY CRICKET AND THE CHIRP LANGUAGE

The Handy Cricket is an inexpensive, hand-held microcontroller that was developed for educational applications. It has been used by teachers and K–12 students, industrial designers, and undergraduates [15, 11, 12]. Figure 1 shows the Cricket controller, and the "Mini-Bot," a small maze-crawling robot built by a 10th grade student.

The device itself includes hardware interfaces for two DC motors and two analog sensors, and is based on a Microchip PIC processor running a custom, stack-based virtual machine. The Cricket's virtual machine supports 16-bit integer numerics, procedure calls (including recursion) with arguments and a return value, looping and conditional control structures, and global variables [10]. Prior to this work, the Cricket was programmed only in a version of Logo, with language features tightly matched to the device's virtual machine.

The Cricket robot controller provides a simple and realistic platform to teach compilers. The hardware control is easy to understand, and the virtual machine uses a small bytecode instruction set. Leveraging the Cricket, we designed a Cricket programming language using the conventional Java/C syntax, which is ideal to teach compiler implementation. This new language is called Chirp.

```
int sum;
int i;
void main()
{
 sum = 0;
 i = 0;
 loop (100) {
   i = i+1;
  sum = sum+i;
 }
 System.Sound.beep();
 System.IR.send(sum);
}
```



```
tree_block
  tree_var_assign
    sum
    0
  tree_var_assign
    i
    0
  tree_loop_count
    100
    tree_block
      tree_var_assign
        i
        +
          tree_var_ref
            i
          1
      tree_var_assign
        sum
        +
          tree_var_ref
            sum
          tree_var_ref
            i
  tree_func_call_stmt
    tree_func_call
      System.Sound.beep
  tree_func_call_stmt
    tree_func_call
      System.IR.send
```

**Figure 2: Example Chirp Program and its Corresponding Abstract Syntax Tree Intermediate Representation (AST IR).**

A sample Chirp program is shown on the left in Figure 2. The program uses a loop to compute sum of 1 to 100. Then it calls system interface functions to produce the alert beep sound, and send the result through Cricket infrared interface.

As noted, the Chirp syntax is similar to C and students with C/C++ and Java experience can quickly learn Chirp.

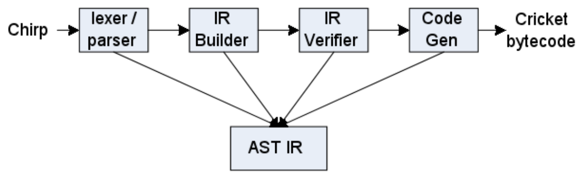


**Figure 3: Chirp Compiler Structure.**

The language features of Chirp are minimal and only include the data types and language constructs that the Cricket robot controller supports:

- **Data types.** Chirp supports two basic data types: integer and byte. Integer types are 16-bit numbers and byte types are 8-bit. Boolean values are similar to C: 0 represents logical false, and any non-zero value represents true. Chirp also supports 1-dimensional arrays of the basic types. There are no pointer or reference types.
- **Variables.** Chirp has only global variables and there are no local variables.
- **Functions, arguments, and statements.** Chirp functions are similar to C functions, which can take formal arguments, and return a result or have void return type. The function body is a sequence of Chirp statements, including assignment, if-else statements, unconditional loops, loops with a calculated loop count, and function returns.
- **Interfaces.** To support robotic control, Chirp provides a special construct—interface. An interface aggregates related I/O functions within a single namespace. Chirp defines built-in System interfaces, which are directly mapped to Cricket bytecodes for motor, sensor and other hardware controls.
- **Expressions.** Chirp supports arithmetic, boolean and relational expressions.

The full language specification of Chirp is documented in [16]. The unique features of Chirp such as interfaces serve the example of idiosyncrasies of real languages. As Chirp has close ties to Cricket bytecode and virtual machine, students will explore how high-level language features are mapped and translated to low-level primitives—the core of compilers/language processor design, with the easy-to-understand Cricket as the target.

## 4. BUILDING THE CHIRP COMPILER

In the following, we describe the construction of Chirp compiler. We follow the general structure of modern compilers [7]. The structure of the Chirp compiler is shown in Figure 3.

### 4.1 Front-end

The front-end consists of Chirp lexer and parser. Students will learn the basics of regular expression and grammar theory and move quickly to use modern compiler tools to build the lexer and parser for Chirp. The compiler tool we use is ANTLR [2]. Students will write specification of Chirp tokens and grammar rules in ANTLR grammar file and ANTLR will generate lexer and parser classes in Java. The simplicity of Chirp and high-level ANTLR specifications make the front-end easy to understand and construct. In our reference implementation, the full spec of Chirp lexer tokens and grammar rules takes about 80 lines. The generated Java lexer and parser classes can then be used directly to build a fully functional front-end.

### 4.2 Intermediate Representation

A key component of modern compilers is the Intermediate Representation (IR), which facilitates program analysis and transformation [7]. The second phase of Chirp compiler builds the IR and carries out semantic checking. We adopt the Abstract Syntax Tree (AST) as the IR; the AST encodes grammar constructs and is a natural representation. The ANTLR tool can automatically construct AST tree nodes when the parser matches a grammar rule. Each rule representing Chirp language constructs such as assignments, if-else conditionals, loops and expressions, is extended with AST-building annotations (IRBuilder). Also, the ANTLR system includes tools to easily create visualizations of the AST data structures (Figure 2).

The Chirp compiler design highlights the central role of IR in modern compilers: in the rest of compiler, the IRVerifier and CodeGen perform tree walks on AST to check semantics and generate bytecode. For example, in IRVerifier, after the AST IR is constructed, the tree walker traverses the tree, checking semantics of each node. During parsing, symbol information is collected into a symbol table, such as type information, variable and function declarations. The IRVerifier tree walker looks up symbol tables and verifies correct and consistent usage of symbols.

### 4.3 Back-end

After IRVerifier runs successfully, the back-end makes a second tree walk on the AST and generates Cricket bytecode. On the Cricket controller, user program data and bytecode instructions are stored in 4KB EEPROM with a linear address space. Chirp's CodeGen tree walker takes three steps to create the bytecode image to run on the Cricket. In Step 1, CodeGen lays out the data for variables and arrays. As Chirp has only global variables, all data storage can be allocated statically. To do so, CodeGen iterates over the symbol table to determine the size and address for variables and arrays. The data storage starts from address 0 and is allocated consecutively. Once the variable is allocated, the corresponding symbol table entry will be updated with its allocated address.

After data allocation is done, CodeGen moves on to Step 2 and generates bytecode instructions for each Chirp function. CodeGen performs a tree walk on the AST IR and generates bytecode for each statement tree node. The Cricket VM uses postfix format for bytecode instructions in which the operands are computed before emitting the operation bytecode. To generate bytecode in postfix order, CodeGen traverses the AST in postorder, generating the bytecode sequence for subtrees first before emitting the bytecode for the tree node.

As variable addresses have been computed in Step 1, the only thing that needs to be fixed after Step 2 is determining the callee address for function calls (this must be deferred until after the bytecode of functions are generated). So in Step 2, for function calls, CodeGen supplies a placeholder

| Week | Topics | Assignments |
|---|---|---|
| 1 | Intro to Cricket bytecode, hardware Cricket, and Virtual Cricket simulator | Manually translate simple Chirp code to bytecode and run on Cricket |
| 2, 3 | Lexer/parser | Use ANTLR to build lexer and parser, generate AST IR |
| 4 | IR verification | Build IRVerifier tree walker; manage symbol tables |
| 5 | Code generation | Build CodeGen tree walker |
| 6 | Testing, and Robot contest | Students use Chirp system to program real robots |

**Table 1: Teaching Schedule of Cricket/Chirp.**

address (2 bytes) for the target function, and records the bytecode and function name in a placeholder list. Then, in Step 3, CodeGen assigns the address for each function and revisits the placeholder list to patch in the final address for function calls. Finally, CodeGen writes the Cricket startup vector with the address of Chirp main function as the program entry point.

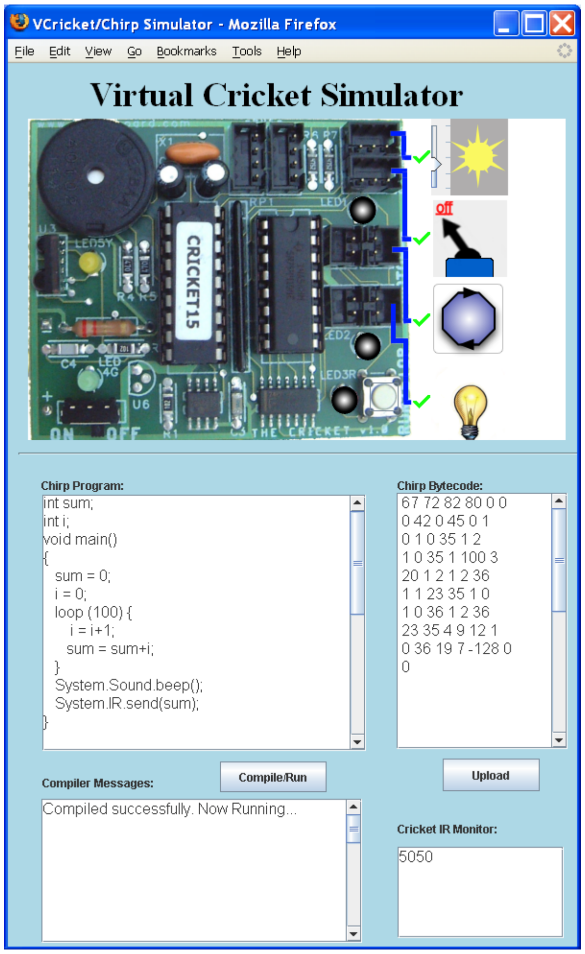
## 5. INTEGRATING CHIRP INTO UNDERGRADUATE PROGRAMMING COURSE

We are integrating teaching the Chirp system into our second year "Computing IV" course (a required core course). Historically, this course has served to introduce students to "programming in the large" (i.e., the design and implementation of a significant project), to reinforce OO concepts and practice, and to introduce advanced development tools. Past versions of the course have used language processing topics including lexing and parsing. In our pedagogical design, we will use Cricket and Chirp framework to teach not only the compiler front-end (lexing and parsing), but also intermediate representation and code generation—important topics often ignored in an introductory course. The Chirp compiler also provides a good implementation target to use OO techniques already taught in first half of the course.

### 5.1 Schedule

We have designed an incremental 6-week schedule to teach the framework. The topics and assignments for each week are listed in Table 1.

For each assignment, students will be given partially implemented Java classes and extend them, implementing more Chirp features. For example, the lexer assignment will give token specs for integers, and students will implement hexadecimal numbers. In the parser assignment, students will be given sample rules for assignment, and implement a more complex loop statement. In the semantic verifier assignment, the given code will check variable references, and students will extend it to arrays. In the codegen assignment, skeleton code emits bytecode for arithmetic expressions, and students will add support for general expressions, function calls and statements. This skeleton code approach will help


VCricket/Chirp Simulator - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Virtual Cricket Simulator
off
Chirp Program:
Chirp Bytecode:
Compiler Messages:
Compile/Run
Upload
Cricket IR Monitor:

```
int sum;
int i;
void main()
{
  sum = 0;
  i = 0;
  loop (100) {
     i = i+1;
    sum = sum+i;
  }
  System.Sound.beep();
  System.IR.send(sum);
}
```

```
67 72 82 80 0 0
0 42 0 45 0 1
0 1 0 35 1 2
1 0 35 1 100 3
20 1 2 1 2 36
1 1 23 35 1 0
1 0 36 1 2 36
23 35 4 9 12 1
0 36 19 7 -128 0
0
```

```
Compiled successfully. Now Running...
```

```
5050
```

**Figure 4: The Virtual Cricket Simulator.**

students navigate through the main topics of compilers without being overwhelmed or distracted.

Our teaching schedule will culminate in a robot competition where students write Chirp programs to control prebuilt Cricket robots, using their compilers to create bytecode and run the robot. This will demonstrate the power and usefulness of compiler techniques they have learned.

### 5.2 Supporting Tools and Techniques

To make the materials easily accessible within the undergraduate programming course, we have developed teaching tools and software techniques to assist students' learning:

- **Virtual Cricket simulator with integrated Chirp compiler.** Previously, Martin *et al.* developed the Virtual Cricket, which allows students to run Cricket code in a web-based emulation of the Cricket hardware. The Virtual Cricket simulates execution of bytecode and shows the actions of motors and sensors with a GUI interface [13].

  To assist students in learning Chirp, we extended this system with the Chirp reference compiler. The combined tool can be run on any Java-enabled web browser (Figure 4). In our design, the simulator is used in two ways:

1. It provides an environment to learn Chirp. The simulator has input window to type in Chirp program; the built-in reference compiler will then compile it to bytecode and run it on the simulator. This helps students learn the language quickly and explore its features.
2. It provides a testing platform. The generated bytecode can both run on Cricket robots and the simulator. The simulator also prints out messages through browser's Java console, so students can check how bytecode is executed during debugging.

- **Compilation tracking.** For students learning compilers first time, they often cite lack of ability to "see" how compiler operates internally as major difficulty in understanding the compilation process [6].

  We use several techniques to help students visualize how a compiler works. First, the ANTLR tool generates a recursive-descent parser. In contrast to the table driven parsers of Yacc and other tools, recursive-descent parsing is intuitive and easier to understand [7]. ANTLR also generates textual tracing messages in the parser to show the top-down parsing process. The tracing option is turned on when building the Chirp compiler. Second, the AST IR can be visualized by the ASTFrame class in ANTLR library. Students can easily generate diagrams of AST trees similar to Figure 2. We expect these visualization techniques will help students to build a mental image of how the compiler works.

- **Error handling.** The use of Java and ANTLR simplifies compiler error handling—an important and difficult topic in compiler design. The ANTLR lexer and parser generates Java exceptions when there are parsing errors. Syntax errors can thus be handled by defining appropriate Java exception handlers. As we use a recursive-descent parser, the error is immediately detected at the grammar rule where the parser failed to match. This gives precise error detection and helps students to design and debug their compilers. Semantic errors in Chirp are handled during AST tree walk by IRVerifier, which checks against symbol tables, and verifies consistent use of variable and function types.

- **OO design using design patterns.** Design patterns have become an essential element in OO programming courses. The Chirp compiler project serves as an illuminating example to teach and apply the commonly used patterns. The IRBuilder, IRVerifier and CodeGen in Figure 3 are implemented using the Singleton pattern to maintain single object instance and global access. The symbol creation is implemented as the Factory pattern. Symbol table and AST tree management use Java generic template classes and are accessed through Iterators. More elaborate patterns such as Visitors can be used in AST tree walkers.

## 6. CONCLUSIONS

Traditionally, compiler construction has been taught as an elective course with limited student attendance. In this paper, we presented an alternative approach to teach compiler fundamentals in an intermediate-level undergraduate programming course.

Leveraging the Cricket embedded robot controller, we designed Chirp—a simple yet realistic language for compiler implementation. The Chirp compiler highlights the techniques and tools in building modern compilers. We also developed the Chirp simulator and used compiler visualization method to assist students learning compilers. The Cricket and Chirp framework provides an accessible and engaging platform to teach compiler fundamentals in the core undergraduate curriculum, so an increased number of students can acquire and benefit from the useful compiler techniques.

We are excited to make the materials described in this paper available to other educators. The Chirp language specification, the Chirp compiler and simulator tools, and related teaching materials are online at `www.cs.uml.edu/∼xu/chirp`. The Handy Cricket hardware is available at `handyboard.com/cricket`.

## 7. REFERENCES

[1] ACM Computing Curricula 2001, Computer Science volume. http://www.sigcse.org/cc2001/.

[2] ANTLR documentation. http://www.antlr.org/doc.

[3] Curriculum '68: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, pages 151–197, Mar. 1968.

[4] Curriculum '78: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, pages 147–166, Mar. 1979.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] K. Andrews, R. R. Henry, and W. K. Yamamoto. Design and implementation of the uw illustrated compiler. In *Proceedings of 1988 conference on Programming Language design and Implementation*.

[7] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2003.

[8] S. Debray. Making compiler design relevant for students who will (most likely) never design a compiler. In *Proceedings of 2002 SIGCSE technical symposium on Computer science education*, pages 341–345.

[9] T. R. Henry. Teaching compiler construction using a domain specific language. In *Proceedings of 2005 SIGCSE technical symposium on Computer science education*, pages 7–11.

[10] F. Martin, B. Mikhak, M. Resnick, B. Silverman, and R. Berg. To mindstorms and beyond: Evolution of a construction kit for magical machines. In A. Druin and J. Hendler, editors, *Robots for Kids: Exploring New Technologies for Learning*, pages 9–33. Morgan Kaufmann, 2000.

[11] F. Martin, B. Mikhak, and B. Silverman. Metacricket: A designer's kit for making computational devices. *IBM Systems Journal*, 39(3 & 4), 2000.

[12] F. G. Martin and S. Kuhn. Computing in context: Integrating an embedded computing project into a course on ethical and societal issues. in press, 2006.

[13] F. G. Martin, N. Palmer, and B. Skinner. The virtual cricket: A web-based simulator for learning robot programming. In preparation.

[14] D. Z. Nguyen, M. Ricken, and S. Wong. Design patterns for parsing. In *Proceedings of 2005 SIGCSE technical symposium on Computer science education*.

[15] M. Resnick, R. Berg, and M. Eisenberg. Beyond black boxes: Bringing transparency and aesthetics back to scientific investigation. *Journal of the Learning Sciences*, 9(1):7–30, 2000.

[16] L. Xu and F. Martin. The chirp language specification. Technical Report TR-2005-003, Dept. of Computer Science, UMass Lowell.