Cricket Chirp: Teaching Compiler Construction/Language Processing Using Embedded Robot Controller

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

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

ABSTRACT

Traditionally, the topic of compiler construction/language processing has been taught as a selective advanced course in Computer Science curricula, due to its rich content and subject complexity. However, under such an approach, students may opt not to take the course and graduate with little understanding and experience with the highly useful techniques embodied in modern compiler construction. In this paper, we present an embedded robot controller-based teaching framework to teach the fundamentals of compiler/language processor in intermediate-level undergraduate programming course.

Our approach leverages the simple yet versatile Cricket educational robot controller. We designed the Chirp language to expose the capabilities of the Cricket virtual machine. Students will build the Chirp compiler and generate Cricket bytecode running on both Chirp simulator and real robot. The Chirp compiler design is organized as a series of components in Java, which demonstrate key compiler construction techniques including lexing, parsing, semantic analysis, error handling and code generation. The Chirp framework provides a balanced and realistic 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, Compilers, Robot Controllers

Keywords

Compiler Construction, Embedded Processor, Educational Robot

1. INTRODUCTION

Compiler Construction has long been included in the computer science curricula. However, due to the rich content and subject complexity, the compiler course is usually relegated as an upper-level selective class. Under such a teaching format, students often opt not to take the course, thus miss the opportunity to acquire the knowledge and skills embodied in a compiler class – 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/language processing into an intermediate-level undergraduate programming course. We leverages the Cricket embedded robot controller and the C-like Chirp language for robot programming, which provide a simple, realistic and engaging platform to teach the basics in compiler design in a core programming course. This compiler teaching framework allows us to present a balanced and accessible coverage on the broadly applicable compiler techniques, including front-end, back-end and intermediate representation in typical compiler development. Software tools and programming techniques are employed to reduce conceptual complexity, so the basics in compiler construction can be effectively taught in a core CS2-level undergraduate class.

The rest of the paper is organized as the following: Section 2 describes the motivation of our approach; Section 3 describes the Cricket embedded robot controller and its programmable interface; Section 4 describes the Chirp language to program cricket robots; Section 5 describes the Chirp compiler design and the approach to teach it in the undergraduate class; Section 6 describs the tools and techniques to ease the learning complexity; Section 7 summarizes related work. Finally we conclude in Section 8.

2. MOTIVATION

Since the early days of computing, Compiler Construction has been a standard offering in the undergraduate Computer Science curricula. However, due to its rich content of theory and practice, and the subject complexity, compiler courses are routinely designated as advanced elective in the curriculum. Both ACM curricula '68 and '78 included Compiler Construction in Computer Science Electives [9, 10]. In ACM Computing Curricula 2001 [5], Compiler Construction is listed as advanced level in the Programming Language block. The widely used compiler textbooks also suggest primary teaching format of dedicated compiler class to cover the main materials, and a second class to teach more ad-

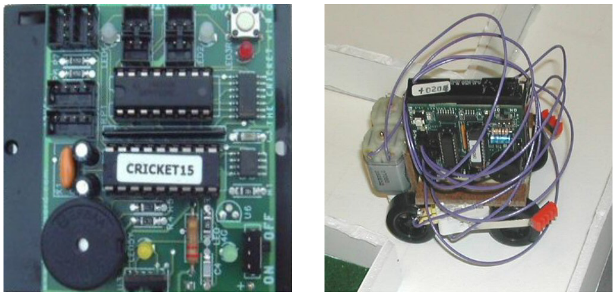


Figure 1: The Cricket controller and maze-crawling Mini-Bot

vanced topics such as program analysis and optimization. This elective class approach, though may provide in-depth coverage of compiler design, however, it often limits the audience to a small number of students, as the majority students have little exposure to compiler topics, and opt not to take the class at all. At UMass Lowell, Compiler Construction is taught as an upper-level selective course and the undergraduate enrollment is routinely one third or less of the size of required core programming courses. Most students graduate without acquiring the techniques and skills provided in the compiler class. In other institutions, educators [6, 4] report similar experiences.

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, and the techniques of language processing can be used to solve many non-compiler related problems [3, 2, 4]. Good knowledge of these topics are essential for students to understand concepts and inner workings of languages and language-based systems. In this sense, Compiler Construction plays similar role as a computer organization class: though not many students will design computers, the class helps them better understand computer systems. As programmable systems become increasingly used in daily lifes, knowledge in fundamentals of compilers is as important to understand and use such systems.

Furthermore, compiler is often the first piece of "real" software a student would write and serves as an illuminating example to apply program design and software engineering techniques, such as design patterns. To realize such benefits and make compiler techniques accessible to students, in this paper, we present a teaching framework to integrate compiler fundamentals into CS2-level intermediate programming course.

3. CRICKET AND CHIRP

As discussed in Section 2, it is desirable to introduce the basics of compilers in the core curriculum. However it is often challenging to do so, due to the large array of potential topics. Often, due to time restriction, only some parts of front-end are covered, such as lexing and parsing, and simple examples like arithmetic expression parsing are used to illustrate principles and the use of parser-generator tools. This approach shows only one piece of the puzzle and

```
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);
}
```

Figure 2: Example Chirp program

students often walk away with the perception that compiler construction is mostly about the grammars and parsing It is therefore important to present the whole picture of modern compiler construction, and to do so, educators must balance the selection of topics and take advantage of compiler building tools and software techniques, so students can grasp the fundamentals effectively without being overwhelmed by the complexity of the subject.

In our compiler teaching framework, we leverage the Cricket embedded robot controller [7] and Chirp language [11] to provide a simple and realistic engaging platform to teach compiler construction. Students will build a working Chirp compiler through a three-phase class project. The compiler will compile robot control programs written in Chirp into bytecode and upload and run on Cricket-driven robots. In this section, we describe the Cricket embedded controller, its virtual machine, and the Chirp language for Cricket.

TODO: summary of Cricket here. (Overview of Cricket. Motors, sensors, virtual machine, bytecode.) Figure 1 shows the Cricket controller and a maze-crawling robot with wheels driven by the two motors and light sensors to detect route in a maze.

Chirp is a simple high-level language designed for programming educational robots, in particular, the Cricket based embedded robots. The Chirp syntax is similar to C and students with C/C++ and Java experiences can quickly learn to use Chirp. However, as Chirp is designed to support robot control, the language features of Chirp have been greatly simplified and only include data types and language constructs esstial for writing cricket control programs.

- 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 values represent true. Chirp also supports single dimension arrays of the basic types. There are no pointer or reference types in Chirp.

- Variables: Chirp has only global variables and there is no function local variables.

- Functions and statements: Chirp functions are similar to C functions, which can take formal arguments, 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 details of Chirp are described in [11]. An example Chirp program is shown in Figure 3, which uses a loop to compute sum of 1 to 100 and call System interface functions to make alert beep sound and send the result through Cricket infrared interface.

Cricket and Chirp platform makes an simple yet realistic and engaging platform to teach fundamentals of compiler design. Chirp is small enough for students to build a compiler, and it represents common language features such as data types, control statements, expressions and functions/procedures. The unique features of Chirp such as interfaces serve the example of idiosynchracy of real languages. 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. Furthermore, the generated code can be directly uploaded and run on Cricket robot or the Chirp simulator (Section 5), giving students an engaging platform to see their compilers into action. The compiler is no longer just about grammar rules and parsing of strings, it is the real force to run embedded robots. This will excite the learning experience and demonstrate the usefulness and power of compiler techniques.

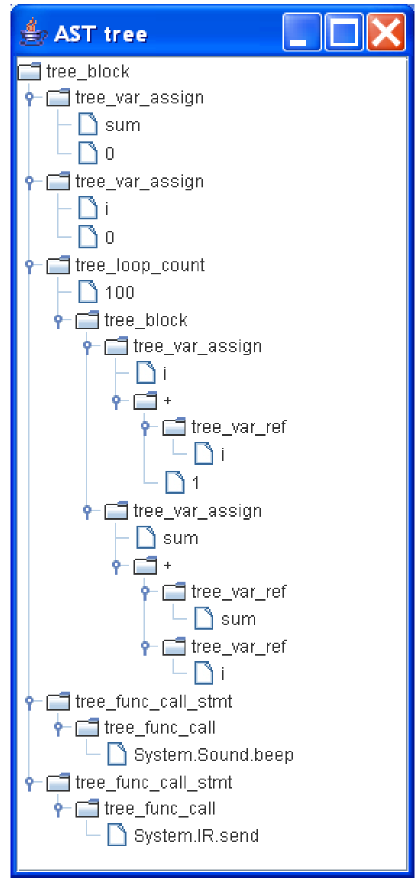


Figure 3: AST Tree IR

4. BUILDING CHIRP COMPILER

Using Cricket and Chirp, we can teach students the core techniques of compilers: the front-end including lexing and parsing, the intermediate representation and semantics checking, and the back-end of code generation. This gives students a complete view of compiler construction, instead of the narrow scope when the focus is limited to parsing or similar topics. Experiences gained in building the Chirp compiler enable students to consolidate knowledge on language and programmable systems and prepare them to use compiler techniques to solve other language-related problems. In the following, we describe the framework to teach the construction of Chirp compiler. We will follow the general structure of modern compilers [3].

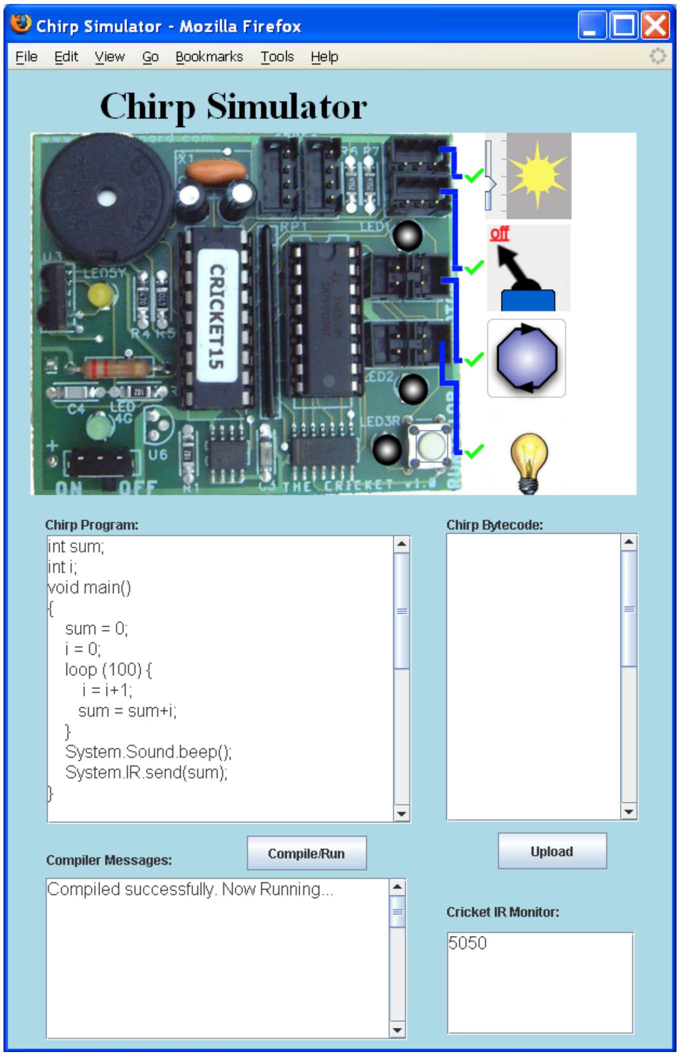
4.1 Front-end

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 [1]. 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

The second phase of Chirp compiler is to build intermediate representation (IR) and carry out semantic checking. A key component of modern compiler is IR representation which facilitates program analysis and transformation. We adopts the abstract syntax tree (AST) as IR representation, which encodes grammar constructs and is conceptully easy to understand by students. A nice feature of ANTLR tool is that it can construct AST tree node when the parser matches a grammar rule. Each rule representing Chirp language constructs such as assignments, if-else conditionals, loops and expressions will be extended with AST tree building annotation. The AST tree representation for the example program in Figure 3 is illustrated in Figure 4.2.

After AST IR is constructed, an AST tree walker will walk down the tree node, checking semantics of each node. During the front-end parsing, symbol information is collected into a symbol table, such as type information, variable and function declarations. The IR Verifier tree walker will look up symbol tables and verify the usage of symbols is correct and consistent.



4.3 Back-end

After IR Verifier runs successfully, the back-end will make a second tree walk on the AST tree and generate Cricket bytecode.

TODO: more details here.

5. ASSISTING TOOLS AND TECHNIQUES

TODO: discuss chirp simulator, ANTLR, Java generics, design patterns here.

6. RELATED WORK

TODO: compare with other compiler teaching work.

Recently, Nguyen et aldescribed teaching recursive- descent parsing using design patterns [8].

7. CONCLUSIONS

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] Antlr documentation. http://www.antlr.org/doc.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2003.

[4] Saumya 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.

[5] CC2001 Task Force. Computing curricula 2001, computer science volume. http://www.sigcse.org/cc2001/, December 2001.

[6] Tyson 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.

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

[8] Dung "Zung" Nguyen, Mathias Ricken, and Stephen Wong. Design patterns for parsing. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, 2005.

[9] ACM Curriculum Committee on Computer Science. Curriculum '68: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, pages 151–197, March 1968.

[10] ACM Curriculum Committee on Computer Science. Curriculum '78: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, pages 147–166, March 1979.

[11] Li Xu and Fred Martin. The chirp language specification. Technical Report TR-2005-003, Dept. of Computer Science, UMass Lowell.