ChocoPy: A Programming Language for Compilers Courses

Rohan Padhye
rohanpadhye@cs.berkeley.edu
University of California, Berkeley
USA

Koushik Sen
ksen@cs.berkeley.edu
University of California, Berkeley
USA

Paul N. Hilfinger
hilfingr@cs.berkeley.edu
University of California, Berkeley
USA

Abstract

ChocoPy is a programming language designed for teaching an undergraduate course on programming languages and compilers. ChocoPy is a restricted subset of Python 3.6, using static type annotations to enforce compile-time type safety. ChocoPy is fully specified using formal grammar, typing rules, and operational semantics. Valid ChocoPy programs can be executed in a standard Python interpreter, producing results consistent with ChocoPy semantics. A major component of CS164 at UC Berkeley is the project: students develop a full compiler for ChocoPy, targeting RISC-V, in about twelve weeks. In other exercises, students extend the syntax, type system, and formal semantics to support additional features of Python. In this paper, we outline (1) the motivations for creating the ChocoPy project, (2) salient features of the language, (3) the resources provided to students to develop their compiler, (4) some insights gained from teaching two semesters of ChocoPy-based courses by different instructors. Our assignment resources are available for re-use by other instructors and institutions.

CCS Concepts • **Social and professional topics** → **Computer science education**; • **Software and its engineering** → *Compilers*; *Context specific languages.*

Keywords Compilers courses, Python, RISC-V

ACM Reference Format:
Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In *Proceedings of the 2019 ACM SIGPLAN SPLASH-E Symposium (SPLASH-E '19), October 25, 2019, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3358711.3361627

SPLASH-E '19, October 25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6989-3/19/10…$15.00
https://doi.org/10.1145/3358711.3361627

1 Introduction

ChocoPy is a programming language designed for classroom use. It is currently used at UC Berkeley to teach CS164, an undergraduate-level course on the design and implementation of programming languages. ChocoPy is a statically typed, restricted subset of Python 3.6. ChocoPy is fully specified using formal descriptions of syntax, typing, and operational semantics. Students taking CS164 learn to reason about such formalisms, consider alternatives, and extend them to support additional features of Python. Over the course of a semester, students work in teams to develop a full compiler for ChocoPy, targeting the RISC-V architecture. We provide to students a language reference manual, a RISC-V implementation guide, and skeleton code for developing a modular ChocoPy compiler in Java. Additional resources include an instructor-provided reference compiler, a web-based ChocoPy IDE, a web-based RISC-V simulator with step-through assembly debugging, and an auto-grader. The ChocoPy compiler project is developed in three modular stages that can be tested and graded independently of each other. The ChocoPy project is designed to be portable. All ChocoPy resources are available at https://chocopy.org.

2 Motivation

Waite [11] describes three common strategies for teaching an undergraduate compilers course: (1) software project, (2) application of theory, (3) support for communicating with a compiler. At UC Berkeley, the CS164 course is a mix of the first two strategies. Students are exposed to various theoretical concepts underpinning programming language design—such as syntax, type systems, and formal semantics—as well as their efficient implementation—such as lexing, parsing, type checking, program analysis and optimization, code generation, and memory management. These concepts concurrently tie in to a semester-long project on developing a full compiler for a small but non-trivial programming language.

For students, developing a compiler end-to-end can be a hugely rewarding task. The rewards are both in the form of a substantial software engineering experience—the project is often the largest that students have undertaken so far—as well as in gaining insights about programming language design and implementation.

The major design decision for instructors is what programming language to base the compiler project on. The

advantage of using a pedagogical language such as COOL [2] or SOOL [4], is the availability of formal specifications for a set of language features designed specifically for education. One of the co-authors of this paper has implemented this approach for several years. However, students are not always motivated to learn about the design of or write a compiler for a language having unfamiliar syntax or whose features are not relatable to languages they already know; this phenomenon has been observed by other instructors as well [1, 11]. An alternate approach is to specify a subset of a popular language, such as Java [3, 6, 8] or C [7]. Another co-author of this paper has had success with this approach in the past. However, existing subset definitions either did not have formal semantics attached to them, were not sufficiently complex enough for reasoning about language design issues, and/or did not have associated resources for developing a compiler targeting a modern instruction-set architecture.

The development of the ChocoPy project was motivated by the following design goals:

1. We wanted to use a subset of a widely used programming language, preferably one which students are already familiar with.
2. We wanted the language to be expressive enough to write non-trivial programs in. In particular, we wanted to support an object-oriented paradigm with sufficient complexity to illustrate important nuances of static type checking and efficient code generation. We decided to use the features of COOL as a reference.
3. We wanted to use a language whose syntax, type-checking rules, and operational semantics were formally specified. These concepts tie the theory component taught in class to practical aspects of compiler development.
4. We wanted the students' compilers to target a modern assembly language, while providing sufficient tool support for simplifying such a daunting task.
5. We wanted to produce artifacts that can be re-used across instructors and institutions offering compilers courses.

ChocoPy fulfills these goals in the following way:

1. ChocoPy is a statically typed subset of Python 3.6. We found that, as of 2018, most students taking CS164 were already familiar with writing Python programs. ChocoPy uses Python's type hinting syntax [5, 10] to annotate variable and formal parameter declarations with static types. All valid ChocoPy programs can also be run in a standard Python interpreter.
2. ChocoPy supports integers, booleans, strings, user-defined classes, lists of any type (including nested lists), class inheritance and method overloading, as well as nested functions that can access non-local variables. Many language features were inspired by COOL, but adapted to conform to Python 3.6. See §3 for more details.
3. The ChocoPy reference manual contains a formal specification of the language's syntax (tokenization rules + grammar), comprehensive typing rules for a type system based on nominal subtyping, as well as operational semantics for all language constructs. Homework exercises lead students towards extending the syntax, typing rules, and formal semantics to support additional Python features such as exceptions, dictionaries, list comprehensions, closures, etc.
4. We provide resources to aid students in implementing a compiler for ChocoPy that targets the 32-bit RISC-V instruction-set architecture [12]. In particular, we provide infrastructure to emit auto-commented assembly code that conforms to conventions listed in an accompanying implementation guide. We also make use of a RISC-V simulator, which allows step-through debugging of RISC-V assembly in a web browser. See §4 for more details.
5. All our artifacts are being made available as a package of three assignments, complete with documentation and auto-graders, for re-use by other instructors upon request.

We soon discovered an additional advantage of basing a compiler project around a type-safe subset of a highly dynamic language such as Python. On non-trivial benchmarks, student-implemented compilers can easily outperform the official Python implementation! We found this to be an excellent source of motivation for students.

3 The ChocoPy Language

ChocoPy is designed to be a subset of Python. An execution of a valid ChocoPy program that does not result in a runtime error should produce the same observable result as the execution of that program in a standard Python interpreter.

Program statements can contain expressions, assignments, and control-flow statements such as conditionals and loops. Evaluation of an expression results in a value that can be an integer, a boolean, a string, an object of a user-defined class, a list, or the special value `None`. ChocoPy does not support dictionaries, first-class functions, and reflective introspection. All expressions are statically typed. Variables (global and local) and class attributes are statically typed, and have only one type throughout their lifetime. Both variables and attributes are explicitly typed using annotations. In function and method definitions, type annotations are used to explicitly specify return type and types of formal parameters.

Figure 1 contains a sample ChocoPy program illustrating top-level functions, statements, global variables, local variables, and type annotations. This is valid Python 3.6 program; the Python interpreter simply ignores the type annotations. In contrast, ChocoPy enforces static type checking

```
def is_zero(items: [int], idx: int) -> bool:
    val: int = 0
    val = items[idx]
    return val == 0

idx: int = 1
print(is_zero([1, 0, 1], idx))
```

Figure 1. ChocoPy program illustrating functions, variables, and static typing. Prints `True` when executed.

```
class Animal(object):
    makes_noise:bool = False

    def make_noise(self: "Animal"):
        if (self.makes_noise):
            print(self.sound())

    def sound(self: "Animal") -> str:
        return "???"

class Cow(Animal):
    def __init__(self: "Cow"):
        self.makes_noise = True

    def sound(self: "Cow") -> str:
        return "moo"

c:Animal = None
c = Cow()
c.make_noise()        # Prints "moo"
```

Figure 2. ChocoPy program illustrating classes, attributes, methods, and inheritance.

at compile time. Figure 2 contains a sample ChocoPy program illustrating classes, attributes, methods, constructors and inheritance.

ChocoPy's syntax is a greatly simplified subset of Python syntax. One huge advantage of this fact is that we get syntax highlighting for free in almost every code editor!

ChocoPy has a nominal type system. The predefined types include `int`, `bool`, `str`, and `object`. Every user-defined class also defines a type. Additionally, for every type `T` in a ChocoPy program, there is a list type `[T]`, which represents a list whose elements are of type `T`. For example, the type `[int]` represents a list of integers. List types are recursive: the type `[[int]]` represents a list whose elements lists of integers.

The semantics of ChocoPy programs have been carefully designed so that the execution of a valid ChocoPy program results in the same observable output as the execution of the same program in a standard Python interpreter.

$$\textsc{var-init}\ \frac{O(id) = T \quad O, M, C, R \vdash e_1 : T_1 \quad T_1 \leq_a T}{O, M, C, R \vdash id{:}T = e_1} \qquad \textsc{attr-init}\ \frac{M(C, id) = T \quad O, M, C, R \vdash e_1 : T_1 \quad T_1 \leq_a T}{O, M, C, R \vdash id{:}T = e_1}$$

$$\textsc{attr-read}\ \frac{O, M, C, R \vdash e_0 : T_0 \quad M(T_0, id) = T}{O, M, C, R \vdash e_0.id : T} \qquad \textsc{list-select}\ \frac{O, M, C, R \vdash e_1 : [T] \quad O, M, C, R \vdash e_2 : int}{O, M, C, R \vdash e_1[e_2] : T}$$

$$\textsc{return-e}\ \frac{O, M, C, R \vdash e : T \quad T \leq_a R}{O, M, C, R \vdash \texttt{return}\ e} \qquad \textsc{return}\ \frac{\texttt{<None>} \leq_a R}{O, M, C, R \vdash \texttt{return}}$$

Figure 3. Sample typing rules for ChocoPy.

$$\textsc{var-read}\ \frac{E(id) = l_{id} \quad S(l_{id}) = v}{G, E, S \vdash id : v, S, \_} \qquad \textsc{var-assign-stmt}\ \frac{G, E, S \vdash e : v, S_1, \_ \quad E(id) = l_{id} \quad S_2 = S_1[v/l_{id}]}{G, E, S \vdash id = e : \_, S_2, \_}$$

$$\textsc{list-select}\ \frac{G, E, S_0 \vdash e_1 : v_1, S_1, \_ \quad G, E, S_1 \vdash e_2 : int(i), S_2, \_ \quad v_1 = [l_1, l_2, \ldots, l_n] \quad 0 \leq i < n \quad v_2 = S_2(l_{i+1})}{G, E, S_0 \vdash e_1[e_2] : v_2, S_2, \_} \qquad \textsc{return-e}\ \frac{G, E, S \vdash e : v, S_1, \_}{G, E, S \vdash \texttt{return}\ e : \_, S_1, v}$$

Figure 4. Sample operational semantics rules for ChocoPy.

The language reference manual[1] provides a comprehensive set of typing rules and operational semantics for all ChocoPy language constructs; some examples are shown in Figure 3 and Figure 4 respectively.

Students learn to read and critique the typing and operational semantics rules. Written exercises walk students through changes or additions to the rules. In particular, they are encouraged to reason about how such changes would affect various other components of the language. Since ChocoPy is a subset of Python, we found it convenient to introduce additional language features that already exist in Python but not in ChocoPy such as dictionaries, exceptions, and default arguments. The official Python documentation and interpreter provide an informal specification and an oracle respectively; these form the basis for constructing and evaluating new formalisms.

[1] https://chocopy.org/chocopy_language_reference.pdf

4 Resources for Compiler Development

To support the compiler development project, students are provided a number of artifacts. The design philosophy for this distribution is strongly influenced by COOL [2].

The main document provided to students is the *ChocoPy language reference manual*. This document formally specifies the language syntax, typing rules, and operational semantics. In the project, students are expected to develop a compiler that conforms to these formal specifications.

The course project consists of three assignments: (1) a front-end that parses ChocoPy programs and produces abstract syntax trees (ASTs), (2) a semantic analyzer and type checker that decorates ASTs with type information, and (3) a code generator that analyzes typed ASTs and emits RISC-V assembly code. Assignments are modular: each of the three compiler stages can be developed and tested independently of the others. Modules from the instructor-provided reference compiler, which is distributed in obfuscated binary form, can be used in place of stages other than the stage being developed. Students can thus test the entire compiler pipeline while working on a single assignment. They need not worry about problems from one of their compiler stages compounding to other stages.

The structure of AST node types is fixed. We use JSON as a seriailized intermediate representation (IR) between the three compiler stages. The output of the first two stages are JSON-serialized ASTs (on success) or JSON-serialized error messages (on failure). Error messages also reference line and column numbers corresponding to the AST node responsible for the error. The schema of the JSON-based IR, which defines the AST node types and error message format, is provided to students along with each assignment specification. The advantage of a serialized IR is that students can freely choose any language of their choice to develop their own compiler. The auto-grader for the first two assignments only compares the JSON output produced by students' implementations against a reference output. The auto-grader for the third assignment simply executes the RISC-V code emitted by the students' compiler and compares it with a reference output (which should also match the output produced by a standard Python interpreter running the same test case).

For each assignment, we provide to students a set of 30–80 small ChocoPy programs and their corresponding reference output. Students can run the auto-grader locally to keep track of their own progress while working on each assignment. The sample test cases provided to students are a subset of the full test suites used to grade their submissions. For each assignment, we also provide Java skeleton code. The skeleton code passes 1 or 2 trivial tests, and demonstrates the basic use of tools and design patterns, e.g. parser generators and AST visitors. Likely owing to the availability of the skeleton code, all student projects to date have opted to implement their compiler in Java.

For the third assignment, which pertains to code generation, we provide to students three additional resources. First, a document called the *RISC-V implementation guide* describes the calling conventions and object memory layout used by the reference compiler. Every student submission to date has followed the same conventions. Second, the reference compiler emits self-documenting assembly code: every line of assembly code emitted for a ChocoPy program contains an associated dynamically generated comment string. The comment explains how the assembly code relates to entities in the ChocoPy program. See Figure 5 for an example. Students seemed to find this useful: the majority of student implementations followed suit and produced their own auto-commented assembly. Third, we provide and host a customized version of the Venus RISC-V simulator [9]. The simulator provides an execution environment for I/O and allows step-through debugging of standalone RISC-V code in a web browser. Registers and memory can be inspected using a GUI. An advantage of using Venus, which itself is written in Kotlin, is that it can be compiled to both JavaScript—for the Web GUI version—and to the JVM—for use by our Java-based auto-grader.

Finally, the ChocoPy website also provides an in-browser IDE. ChocoPy programs can be edited with syntax highlighting and compiled to RISC-V assembly. The assembly can in turn be executed in the same browser window using Venus. Compile-time errors highlight mistakes in the source code. See Figure 6 for an example. The IDE is a recent development. Since the students' compilers produce JSON-serialized error messages with line and column numbers in a standard format, it is possible to integrate their own implementations into this IDE. We did not have this integration available in past offerings but intend to provide such a mechanism in the future.

5 Informal Survey

Mid-way through the Fall 2018 offering of CS164, we conducted an informal, anonymous survey of students taking the class. This was at the point where the second programming assignment—on semantic analysis and type checking—was due, but the third assignment—on code generation—was not yet published. The purpose of the survey was for instructors to get feedback on how the students were coping with the then brand-new project, and possibly implement changes in "real-time". We discuss some results here simply as anecdotes and not as a formal evaluation of our work.

The survey listed six negative and two positive statements about the project; students could check any statements they agreed with. There were also text boxes for students to write what they liked about the project, what they did not like, and what they would like changed. Of the 15 survey respondents, 14 checked the positive sentiment "I'm getting insights into how real compilers work", while 9 indicated

```
.globl $is_zero
$is_zero:
  la a0, const_2          # Initial value for local: val
  sw a0, 0(sp)            # Put local variable on stack top + 0
  sw fp, -4(sp)           # Put control link on stack top + 1
  sw ra, -8(sp)           # Put return address on stack top + 2
  addi sp, sp, -12        # Increment stack pointer by 3
  addi fp, sp, 4          # New fp is just below stack top
  lw a0, 16(fp)           # Load var: is_zero.items
  sw a0, -4(fp)           # Push on stack slot 1
  lw a0, 12(fp)           # Load var: is_zero.idx
```

Figure 5. Self-documenting assembly code generated by the ChocoPy reference compiler. Code corresponds to the beginning of function `is_zero` defined in Fig. 1. The label `const_2` references the integer object for value `0`.

```
def is_zero(items: [int], idx: bool) -> bool:
    val: int = 0
    val = items[idx]
    Index is of non-integer type `bool`
idx: int = 1
print(is_zero([1, 0, 1], idx))
```

Figure 6. Snapshot of the Web-based ChocoPy IDE with syntax and error highlighting. The above program contains a slight change to the example in Fig. 1, which leads to two compile-time type errors.

that they loved ChocoPy. Zero respondents checked the negative statement "I don't like ChocoPy". The most common negative sentiments were "Too much work to do" and "Too much text to read" (6 of 15 respondents each). 4 respondents checked "Implementing error reporting is annoying". Such feedback prompted us to augment the final assignment—on code generation—with a substantial amount of support code to provide to students. Further, the error-reporting requirements for the compiler front-end were streamlined for the second offering of the course (Spring 2019). Six of the seven students who filled out the text boxes indicated either that they found the project well-designed or that they felt they are learning a lot from it.

6 Conclusion

Overall, student feedback about the ChocoPy-based course has been largely positive. This has encouraged us to continue using and refining the language and project materials for future offerings of CS164. The ChocoPy project materials are self-contained and portable. We believe that it should also be easy to customize the project by adding or removing language features as found necessary by instructors. The resources that we developed for this project are available for use by other instructors and institutions, upon request.

Acknowledgments

We thank Rohan Bavishi for aiding in the migration from COOL. We also thank the students of CS164 Fall 2018 for their invaluable feedback on the first deployment of the ChocoPy project. This work is supported in part by NSF grants CCF-1409872, CCF-1908870, CCF-1900968, and CNS-1817122.

References

[1] Alfred V. Aho. 2008. Teaching the Compilers Course. *SIGCSE Bull.* 40, 4 (Nov. 2008), 6–8. https://doi.org/10.1145/1473195.1473196

[2] Alexander Aiken. 1996. Cool: A Portable Project for Teaching Compiler Construction. *SIGPLAN Not.* 31, 7 (July 1996), 19–24. https://doi.org/10.1145/381841.381847

[3] Andrew W. Appel and Jens Palsberg. 2003. *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press.

[4] Kim B Bruce. 2002. SOOL, a Simple Object-Oriented Language. In *Foundations of object-oriented languages: types and semantics.* MIT press, Cambridge, Massachusetts, Chapter 10, 173–200.

[5] Ryan Gonzalez, Philip House, Ivan Levkivskyi, Lisa Roach, and Guido van Rossum. 2016. PEP 526 – Syntax for Variable Annotations. https://www.python.org/dev/peps/pep-0526/.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450.

[7] Christoph M. Kirsch. 2017. Selfie and the Basics. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017).* 198–213. https://doi.org/10.1145/3133850.3133857

[8] Eric Roberts. 2001. An Overview of MiniJava. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01).* ACM, 1–5. https://doi.org/10.1145/364447.364525

[9] Keyhan Vakil. 2017. Venus: RISC-V instruction set simulator built for education. https://github.com/kvakil/venus Retrieved Sept 1, 2018.

[10] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP 484 – Type Hints. https://www.python.org/dev/peps/pep-0484/.

[11] William M. Waite. 2006. The Compiler Course in Today's Curriculum: Three Strategies. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06).* ACM, New York, NY, USA, 87–91. https://doi.org/10.1145/1121341.1121371

[12] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA Version 2.0.* Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley.