ChocoPy compiler

Tobias Karlsson
Computer Science, Lund University, Sweden
to2525ka-s@student.lu.se

Abstract

The purpose of this essay is to investigate the possibility of using ChocoPy as the programming language when educating students on how to construct a compiler as well as comparing the performance of the ChocoPy compiler with a python compiler. The compiler for ChocoPy is implemented to generate assembly code, targeting X86. ChocoPy could be advantageous to use when teaching compilers since it is a simplified subset of Python. When comparisons were made between the ChocoPy compiler and python interpreter for a certain type of program, the result was that the ChocoPy compiler was between 10 and 20 times faster, depending on the size of the program. A ChocoPy compiler can be created in a compiler course since the resources and knowledge provided by the course is enough to handle all the problems that may arise. Given that the resources and time aspect is enough as well as the fact that the ChocoPy compiler is faster than the python interpreter, ChocoPy would be an advantageous programming language to use in a compiler course, given that the objective of the course is to construct a compiler.

1 Introduction

In the process of teaching how to construct a compiler, there are a few things that are important. Since there is a choice what language to choose for the compiler construction, it is of great importance that the language that the compiler is to compile is small and simple enough so that the compiler can be implemented by the students within the time of the course as well as with the tools provided by the course [6]. The purpose of this essay is to investigate if there is a possibility for ChocoPy to be advantageous in the teaching process of constructing compilers as well as comparing the performance of the ChocoPy compiler with a python compiler. The essay will also act as a guide through the creation process of a compiler for the programming language, ChocoPy. The difference between this compiler and earlier projects, done at Berkley is that the toolchain used in this compiler is the same as the one from the compiler course at LTH, which is different from the one that is used at Berkley. One additional difference is that it targets x86 assembly code instead of RISC-V, which is used at Berkley [4].

Course paper, EDAN70, Lund University, Sweden
January 26, 2021.

ChocoPy has big similarities with Python, it can be described as a simplified version of Python [5]. This big similarity and simplification of the Python language could mean that some of the more advanced aspects of a Python compiler could be implemented during the duration of a course.

The compiler for ChocoPy, constructed in this essay is implemented to generate assembly code, targeting X86. ChocoPy language is fully specified with both formal grammar and semantic rules [4]. In the construction of this compiler, JastAdd will be used. JastAdd is a system that implements reference attribute grammar. Creating a compiler for an unknown but still highly relevant language, like ChocoPy will increase the total understanding of how programming languages work and why they are constructed in such a strict grammar. Knowledge will also be obtained about the way compilers handle the different languages and their respective problems, as well as how to use JastAdd advantageously. One example of a common issue for compilers is the indentation syntax that is used in python, instead of the standard brackets that we know from Java. When comparisons were made between the ChocoPy and python compilers for certain programs the result was that the ChocoPy compiler was between 10 and 20 times faster, depending on the size of the program.

Compared to other programming languages, ChocoPy has some unusual problems. These problems could be beneficial to handle when it comes to the teaching process. These additional problems can be solved with the knowledge that is given during a compiler course. Given the compiler comparison, it would also be beneficial for the students to construct a compiler that is faster than big corporate compilers, like the python compiler. Due to the benefits that the student can receive from constructing a ChocoPy compiler and the fact that the resource and time aspect is enough, ChocoPy would be an advantageous programming language to use in a compiler course, given that the objective of the course is to construct a compiler. ChocoPy could, therefore, advantageously be used when teaching the students how to construct compilers.

2 Background

2.1 ChocoPy

ChocoPy is intended to be a subset of python [4]. It is designed to fit into a classroom setting so that students can, under a period of 12 weeks, implement a compiler. Since ChocoPy is designed in such a similar way to Python, every

ChocoPy program is meant to be a valid Python program [4]. The syntax is very similar to python but one important aspect to know is that a ChocoPy program is only contained in one file. A ChocoPy program contains one or more of the following definitions, variable definitions, function definitions and class definitions. The definitions are followed by one or more statements.

2.2 Off-side rule

A language is said to follow the off-side rule if the blocks are expressed by indentation and dedentation [1]. This rule is used by some programming languages, for example, Python. The languages that are using this rule, have a problem with noticing and parsing indentation and dedentation as well as keeping track of previous indentation levels.

3 Compiler construction

The compiler was constructed in such a way that it first targeted a small part of ChocoPy. As the project continued and the smaller parts of the language were completed, additional functionalities of the language were added. A compiler usually consists of lexical analysis, parsing, abstract syntax, semantic analysis and code generation [2]. The steps that were made in this project is displayed below. There are more steps to making a compiler, but the steps below are the most crucial.

3.1 Pipeline

This subsection will work as an overview of the basic Pipeline of this compiler. As seen in figure 1, the compilation start by sending the input file to the preprocessor. The input file is sent to the preprocessor first since the scanner have a hard time handling indentation, more on this in section 2. The preprocessor simplifies the input file by adding indentation tokens. The next step is to send the file to the scanner which adds specified tokens and creates an output of tokens. After the scanner, the file is sent to the parser which uses the grammar to see if the syntax is correct. If the syntax is correct, the output of this stage is an Abstract syntax tree.

When the syntax is correct the code is sent to semantic analysis where Jastadd reference attribute grammars are used. As an example, the semantic analysis check that assigned values to variables is the expected type [3]. The semantic analysis, using reference attributes, creates an attributed abstract syntax tree [3]. After this, the file is sent to the assembly code generator which follows the grammar and produces the correct x86 assembly code. As seen in figure 2.

3.2 PreProcessor

ChocoPy uses indentation to indicate that there is a block of code. The indentation consists of spaces and is complicated to notice directly from the scanner. There is one main reason why indentation is hard to handle from the scanner. The reason is that whitespace should be ignored in some parts of the program, it's, therefore, difficult to count and translate whitespace into indentations since it is being ignored at the same time. It is of big importance to notice the indentation and dedentation since ChocoPy follows the off-side rule. To make the indentation and dedentation obvious for the scanner, the chocopy program will go through the Preprocessor first which constructs a new file with tokens for indentation and dedentation as shown in the figure: 2. The pre-processor has one attribute that keeps hold of the previous indentation level and one attribute that contains the local indentation level. The inner workings of the preprocessor are shown below in step form for clarity where input file preparation contains a description in how the file is prepared and handled by the preprocessor and the looping stage describes the inner workings of the preprocessor.

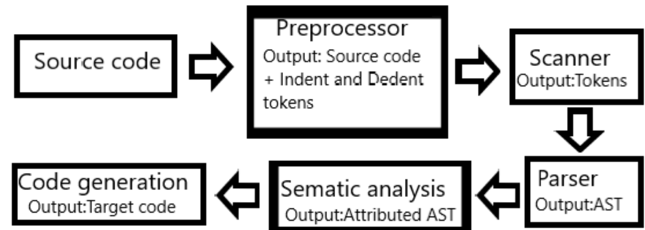

Figure 1. Pipeline

3.2.1 Input file preparation

The first step is that the pre-processor receives the ChocoPy program as an input file. This file is translated into lines of strings, where each line represents a line of the input program. The lines will be looped through until the end of the program is reached.

3.2.2 Looping stage

The pre-processor will loop through every character of the respective line, but only if the character is represented as space. When the processor notices a character that is not space, it will translate the number of spaces into a certain number of indentations or dedentations. The preprocessor keeps a count of the last indentation level. The knowledge of the last indentation level and the current number of spaces will allow the preprocessor to either add a dedentation token or an indentation token. After this, it will prepare to terminate and continue to the next line and repeat the process.

3.2.3 Example

One example of this is the program that can be seen in figure: 2, containing one function definition with a while loop and an if statement. Before the if statement there is a dedentation token, this is since the preprocessor calculated out that the count integer is zero for the current line and that the

previous indentation level was one. The line under the if statement starts with an indentation token, this is since the count integer for that line is 4 and the previous indentation level is one. More details about the preprocessor can be seen when studying figure: 2.

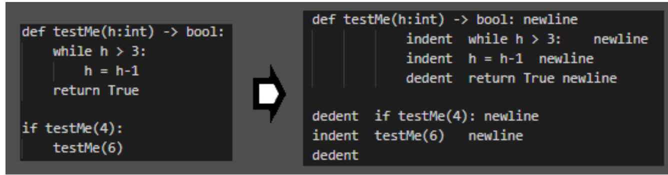

Figure 2. Explaining image of preprocesor

3.3 Scanner

The first step that was made in the process of creating the scanner was to decide what basic tokens should be included in the compiler. This was done by first understanding the ChocoPy language and stating the basic tokens. More tokens are added as the language expands. One example of a token is the if token which tells the scanner to return an IF-token. This token is used in the parser to notice the start of an if statement. The main problem that occurred with the creation of the scanner was to notice indentations or dedentation and return a token for that, this problem was solved by introducing the preprocessor that has control over the dedentation and indentation.

3.4 Parser

The construction of the parser for ChocoPy is much like the construction of the scanner in the way that it starts small and then expands. One example of a row of code in the parser is the following below.

functioncall= idcexpr.id LPAREN parameters.e RPAREN

block= NEWLINE INDENT stmt.s stmtlist.stl DEDENT

The first example above checks for a function call that starts with an id, followed by a left parenthesis, parameters, and right parentheses. The second example shows the usage of indent and dedent, there must be an indent at the start of the block and a dedent token at the end. If any of the tokens are wrong, it cannot be matched with a function call or a block.

3.5 Nameanalysis

Name analysis is needed in this stage of the compiler since it is crucial to check that everything is done according to the rules of the language. For example, it is important to link all uses of variables to a variable declaration. This is important since the compiler has to know whether the variable is declared or not, if it is not declared it should not be accepted to use it. Another example of when the analysis is needed is when variables are declared, there must be a way to check so that there is not already a declared variable with the same name as the current. This check is done by using a lookup pattern [3], which looks through the relevant part of the program.

3.6 Typeanalysis

Type analysis is needed in this stage since it is of great importance to check that the types used are correct. For example, it is required to check that an integer variable is not assigned to a boolean value. This was implemented by introducing two attributes for each variable declaration and variable assignment, these attributes are type and expected type. The attribute type keeps track of what type is being assigned to the variable and the expected type is the type that the variable has. For a better understanding, the example below is provided, the variable declaration has the type:boolean and expected type:int.

$$x : int = \text{True}$$

3.7 Code generation

The main problem with the code generation was to keep track of variable addresses and where to store their values. This problem was solved by introducing an address attribute for each Id declaration. The address depends on the local index of the declaration and if the declaration is a parameter or not. In the outer level of the program, the local index depends on previous variables address, where the first index is set and in functions, the index of the local variable depends on the quantity and order of the variables, within the function.

4 Implemented part of ChocoPy

In this section, there will be a table showing what is implemented in the ChocoPy language and clarification of major parts that are not implemented, 4. The big part from the specification of ChocoPy that is not included is some small things in the class definitions and the array-type. Function definitions are implemented, as well as while loops and if statements which give the user a chance to run function based programs. Besides the array type, all other types are implemented. All different binary operations are implemented as well as literals. The print functionality is implemented as a base function and does not have to be declared before use.

var def	func def	class def, lists	func call
✓	✓	✓+ ✗	✓
if-stmt	while-stmt	return-stmt	var-assignment
✓	✓	✓	✓
for-stmt	pass-stmt	binary-operations	literals
✓	✓	✓	✓

For a better understanding of what is implemented in the compiler, there is one example provided in the following

figure: 3. This example shows a large part of what can be done with the implemented language.

```
e:TestOne = TestOne(True)

class TestOne(object):
    nbr:int=9
    nbr2:int=10
    testB:bool=False

    def __init__ (s:bool) :
        t=s
    def getnbr () -> int:
        if testB:
            return nbr2
        return nbr
    def getnbr12 () -> int:
        return nbr*nbr2
    def incnbr (f:int):
        m:int=5
        nbr=nbr*m*f
        return
def testMe(h:int) -> bool:
    if h>3:
        testMe(h-1)
        print(h*h)
    return True

print(e.getnbr())
e.incnbr(7)
print(e.getnbr12())
testMe(inp)
```

Figure 3. Example of program

5 Evaluation

In this section, an evaluation will be done regarding the possibility of using the construction process of a ChocoPy compiler when teching a compiler course. There will also be a performance comparison between the Python compilator and the ChocoPy compiler, this is a possible comparison since most ChocoPy programs are compatible as python program aswell.

5.1 Teaching compilers with ChocoPy

The possibility of using ChocoPy when teaching the inner workings of creating an compiler has been evaluated with two main criterias. The first criteria is if it is possible to construct a compiler for ChocoPy during the time of the course. The second criterion is whether the knowledge gained from the course and with the help that can be obtained from teachers is enough to create a ChocoPy compiler. For clarity this will be shown as subsections below.

5.1.1 Time criteria

The evaluation of the time criterion is made based on the time it took to develop the ChocoPy compiler for this project, during the time of a project course. The result is that there was not enough time to develop the compiler for the whole language of ChocoPy. The time was not enough to create the complete compiler during the course, one reason for this is since there were also other elements of the course and since there were some parts where I did not know enough about. When evaluating the possibility for a compiler course to make usage of the ChocoPy language, it was concluded that the combination of a more structured and controlled work structure, as well as more allocated time for the construction process, would be sufficient to construct the complete compiler for ChocoPy. The conclusion is that time criteria would not go against the use of ChocoPy.

5.1.2 Knowledge criteria

The evaluation of the knowledge criteria is done during the same timeframe as the time criteria, during the construction of the ChocoPy compiler. The knowledge gained from a compiler course is enough to create a ChocoPy compiler but there were some parts of the construction process for a ChocoPy compiler that was difficult and time-demanding to complete without examples to learn from. When evaluating the possibility for a compiler course, the knowledge obtained by the students would be enough to deal with a ChocoPy compiler since there will also exist examples and clear instruction from the teachers on each part of creating the compiler.

5.2 Comparison with the Python compiler

Comparisons made of the performance by a Python and ChocoPy compiler was made with different types of programs, which were timed. There was one type of program that was tested in more detail, since the time complexity of the program is high for the Python interpreter, the program used in the testing was nested while-loops with a variable assignment and function call. The while-loop was tested with a different number of loops. The results are shown in table 1, the results are shown for six different values of the number of loops for each loop, represented as n where the time complexity is O(n^2).

n	Runtime Python(s)	Runtime ChocoPy(s)
$9*10^3$	10.04	1.02
$12*10^3$	19.56	1.43
$15*10^3$	32.74	1.83
$20*10^3$	61.56	2.83
$25*10^3$	95.63	4.12
$30*10^3$	115.31	5.91

Table 1. Python vs ChocoPy compiler

From inspecting the table above, it can be concluded that the execution time for certain programs is much smaller in the ChocoPy compiler than the Python compiler.

The same type of comparison is done with a certain number of function calls to see which compiler that can do it the fastest. The function that is called only perform one additional operation. The results are shown in table 2, where n is number of calls. n will be big so that the real difference can be shown.

n	Runtime Python(s)	Runtime ChocoPy(s)
8 $*10^7$	5.8	0.49
10 $*10^7$	7.3	0.5
20 $*10^7$	16.12	0.55
50 $*10^7$	41.75	0.7
80 $*10^7$	60.1	0.88

Table 2. Python vs ChocoPy compiler

6 Conclusion

I have started the development of a ChocoPy compiler with the toolchain that is used in the compiler course at LTH. I have evaluated whether the ChocoPy language could be beneficial to use in a compiler course, where the time and knowledge aspect was in focus. I have also evaluated the performance of the ChocoPy compiler versus the performance of the Python compiler. The performance evaluation was done by comparing the runtime of the same program, executed by either the ChocoPy compiler or the Python compiler. In the evaluation done in section 5.2, the Chocopy compiler was between (Look at)5.4 and 104.54 times faster than the Python compiler for a program consisting of nested while-loops. From the comparison result, I concluded that one additional reason to work with ChocoPy in a compiler course is that it could be a beneficial aspect of the course that the students learn that they can, in the span of just one course, create a compiler that is in some ways more effective than the Python compiler.

The result of the time and knowledge aspect, evaluated in section 5.1.2 and 5.1.1 is that the period that a compiler course runs through would be sufficient to implement the whole language.
I have concluded that ChocoPy can be used to advantage in a compiler course if the following points below are met.

· The creation process of the compiler should be in a lab form with collaboration between two students in each group.
· there must be examples for each part of the creation process of the compiler. These examples should be from a compiler that implements a language that is similar to ChocoPy.
· The compiler course should have an additional focus on object-oriented languages and how to generate code for these.

Future work is needed before the ChocoPy language could be adapted within a compiler course. A minimal language would have to be created, this language should be similar to ChocoPy but smaller. There must also be examples for every part of the compiler constructions phase, using the ChocoPy-alike language created for this course.

Acknowledgments

Feedback and guidence from Alfred Åkesson.

References

[1] Michael Adams. 2013. Principled Parsing for Indentation-Sensitive Languages Revisiting Landin's Offside Rule. *ACM SIGPLAN Notices* 48, 1–12. https://michaeldadams.org/papers/layout_parsing/LayoutParsing.pdf
[2] Andrew W. Appel and Jens Palsberg. 2002. *Modern Compiler Implementation in Java.*
[3] Görel Hedin Niklas Fors and Emma Söderberg. 2020. Principles and Patterns of JastAdd-Style Reference Attribute Grammars. (2020), 86–100. https://dl.acm.org/doi/pdf/10.1145/3426425.3426934
[4] Rohan Padhye and Koushik Sen. 2019. ChocoPy v2.2: Language Manual and Reference. (2019), 127–145. https://chocopy.org/chocopy_language_reference.pdf
[5] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2007. ChocoPy: A Programming Language for Compilers Courses. In *ACM SIGPLAN SPLASH-E Symposium SPLASH-E '19, October 25, 2019.* ACM, New York, USA, 1–5. https://doi.org/10.1145/3358711.3361627
[6] Alfred V.Aho. 2008. Teaching the Compilers Course. *ACM SIGCSE Bulletin* 1, 1–4. https://doi.org/10.1145/1473195.1473196