ChocoPy v0.1: Language Manual and Reference

Designed by Rohan Padhye and Koushik Sen

University of California, Berkeley

September 24, 2018

Contents

1 Introduction **3**

2 A tour of ChocoPy **3**
2.1 Functions, statements, and variables . . . 3
2.2 Classes, attributes, methods, and inheritance . . . 4
2.3 Types . . . 5
2.4 Values . . . 5
2.4.1 Integers . . . 5
2.4.2 Strings . . . 5
2.4.3 Lists . . . 5
2.4.4 Objects of user-defined classes . . . 6
2.4.5 `None` . . . 6
2.5 Variable declarations . . . 6
2.6 Expressions . . . 6
2.6.1 Literals and identifiers . . . 6
2.6.2 Arithmetic expressions . . . 6
2.6.3 Logical expressions . . . 7
2.6.4 Relational expressions . . . 7
2.6.5 Concatenation expressions . . . 7
2.6.6 Access expressions . . . 7
2.6.7 Call expressions . . . 7
2.7 Statements . . . 7
2.7.1 Control flow . . . 7
2.7.2 Expression statements . . . 8
2.7.3 Compound statements: conditionals and loops . . . 8
2.7.4 Assignment statements . . . 9
2.7.5 Other statements . . . 9
2.7.6 Predefined classes and functions . . . 9

3 Lexical structure **9**
3.1 Line structure . . . 10
3.1.1 Physical lines . . . 10
3.1.2 Logical lines . . . 10
3.1.3 Comments . . . 10
3.1.4 Blank lines . . . 10
3.1.5 Indentation . . . 10
3.1.6 Whitespace between tokens . . . 11
3.2 Identifiers . . . 11
3.3 Keywords . . . 11

3.4 Literals . . . 11
3.4.1 String literals . . . 11
3.4.2 Integer literals . . . 11
3.5 Operators and delimiters . . . 12

4 Syntax **12**
4.1 Precedence . . . 12

5 Type system **12**

6 Semantics **12**

7 Acknowledgements **12**

```
1 def is_zero(items: [int], idx: int) -> bool:
2     val:int = 0              # Type is explicitly declared
3     val = items[idx]
4     return val == 0
5
6 mylist: [int] = None
7 mylist = [1, 0, 1]
8 print(is_zero(mylist, 1))   # Prints 'True'
```

Figure 1: ChocoPy program illustrating functions, variables, and static typing.

1 Introduction

This manual describes the ChocoPy language, which is a statically typed subset of Python 3.6. ChocoPy is intended to be used in a classroom setting. It has been designed to be small enough for students to implement a full ChocoPy compiler over one semester.

ChocoPy is a subset of Python. Specifically, any valid ChocoPy program is also a valid Python 3.6 program. An execution of a ChocoPy program that does not result in error has the same observable semantics as the execution of that program in Python 3.6.

2 A tour of ChocoPy

Each ChocoPy program must be contained in a single source file. At the top level, a ChocoPy program consists of a sequence of variable definitions, function definitions, and class definitions followed by a sequence of statements. A class consists of a sequence of attributes definitions and method definitions. Functions can be defined inside other methods and functions. A class creates a user-defined type. All class names and functions defined at the top level are globally visible. Class, function, and variable definitions can be interleaved with each other. Classes, functions, and methods cannot be redefined. Program statements can contain expressions, assignments, and control-flow statements such as conditionals and loops. Evaluation of an expression results in a value which can be an integer, a boolean, a string, an object of 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.

2.1 Functions, statements, and variables

Figure 1 contains a sample program in ChocoPy that exposes some basic features of the language. A function can appear at the top-level of a program or could be nested inside other functions or methods. A function may have zero or more parameters which are explicitly typed as is done for the parameters `items` and `idx`. A function also declares its return type. For example, the return type of the function `is_zero` is `bool`. The identifiers used in formal parameters must be distinct and cannot be same as any local variable identifier declared in the body of the function. A formal parameter hides any definition of global or nonlocal variable with the same name. The type of a function body must conform to the declared return type of the function. Function `is_zero` is defined at the top level, and invoked from a top-level statement at line 8. A function can define local variables. A variable definition must declare the type of the variable and must initialize the variable with a literal. The type annotations shown in Figure 1 are valid syntaxes in Python 3.6, though the Python interpreter simply ignores these annotations and uses them as hints for other tools. In contrast, ChocoPy enforces static type checking at compile time.

```
1  class animal(object):
2      makes_noise:bool = False
3
4      def make_noise(self: "animal") -> object:
5          if (self.makes_noise):
6              print(self.sound())
7
8      def sound(self: "animal") -> str:
9          return "???"
10
11 class cow(animal):
12     def __init__(self: "cow", noisy: bool) -> object:
13         self.makes_noise = noisy
14
15     def sound(self: "cow") -> str:
16         return "moo"
17
18 c:animal = None
19 c = cow(True)
20 c.make_noise()       # Prints "moo"
```

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

2.2 Classes, attributes, methods, and inheritance

Figure 2 contains a ChocoPy program that defines two classes: `animal` and `cow`. A class must inherit from another class, called the super-class. For example, `cow` inherits from `animal`, which in turn inherits from the predefined root class `object`. If a class `C` inherits `P`, then `C` is the subtype of `P`, `P` is called the parent of `C`, and `C` is a child of `P`. The semantics of `C` inherits `P` is that `C` has all of the attributes and methods defined in `P` in addition to its own attributes and methods. A class inherits only from a single class; this is aptly called "single inheritance." The parent-child relation on classes defines a graph. This graph cannot contain cycles. For example, if `C` inherits from `P`, then `P` must not inherit from `C`. Furthermore, if `C` inherits from `P`, then `P` must have a class definition somewhere in the program. Because ChocoPy has single inheritance, it follows that if both of these restrictions are satisfied, then the inheritance graph forms a tree with `object` as the root. In addition to `object`, ChocoPy has four other pre-defined classes: `int`, `string`, `bool`, and `list`.

Classes can define attributes, such as the boolean attribute `makes_noise` in `animal`. Like variables, attributes are explicitly typed using type annotations and are initialized with literals. Attributes are inherited by subclasses, which is why class `cow` can access the attribute `makes_noise` at line 13. Attributes cannot be re-declared in a subclass.

Classes can also define methods. Method definitions are exactly same as function definitions with one exception: a method definition must contain at least one formal parameter; the first parameter is bound to the receiver object on which the method is invoked. Methods are inherited by subclasses unless they are overridden, as is done by the method `sound` of class `cow`. To ensure type safety, there are restrictions on the redefinition of inherited methods. The rule is simple: if a class `C` inherits a method `f` from an ancestor class `P`, then `C` may override the inherited definition of `f` provided the number of arguments, the types of the formal parameters, and the return type are exactly the same in both definitions.

In order to create an object `o` of type `C`, the class `C` is treated as a global function and is called with suitable arguments. During the invocation, `__init__` method in the class `C` is invoked and the arguments are bound to the formal parameters of the `__init__` method. If `__init__` method is not defined in class `C`, then the inherited `__init__` method is called. The root class `object` has a default `__init__` method whose body is empty. In Figure 2, the constructor for class `cow` is invoked with argument `True` at line 19.

2.3 Types

Type annotations can be provided in one of two forms: as identifiers or as string literal. In ChocoPy, one could use any of the two forms for annotations. However, in Python the former form cannot be used when defining an attribute that is of the same type as the enclosing class, since the name may not have been bound to the class until the entire class definition has been processed. Since we want ChocoPy to behave similarly as Python, we will use the latter form of annotation in the above described scenario.

In ChocoPy, every class name is also a type. The basic type rule in ChocoPy is that if a method or variable expects a value of type `P`, then any value of type `C` may be used instead, provided that `P` is an ancestor of `C` in the class hierarchy. In other words, if `C` inherits from `P`, either directly or indirectly, then a `C` can be used wherever a `P` would suffice. When an object of class `C` may be used in place of an object of class `P`, we say that `C` conforms to `P` or that $\texttt{C} \leq \texttt{P}$ (think: `C` is lower down in the inheritance tree). As discussed above, conformance is defined in terms of the inheritance graph. Let `A`, `C`, and `P` be types. Then conformance (i.e. $\leq$) is defined as follows:

- $\texttt{A} \leq \texttt{A}$ for all types `A`
- if `C` inherits from `P`, then $\texttt{C} \leq \texttt{P}$
- if $\texttt{A} \leq \texttt{C}$ and $\texttt{C} \leq \texttt{P}$, then $\texttt{A} \leq \texttt{P}$

ChocoPy has a root type `object`, and the predefined types `int`, `bool`, `str`, and `list` as subclasses of `object`. Because `object` is the root of the class hierarchy, it follows that $\texttt{A} \leq \texttt{object}$ for all types `A`.

2.4 Values

In ChocoPy, we can have the following kinds of values.

2.4.1 Integers

Integers are signed and are represented using 32 bits. The range of integers is from -2^{31} to $(2^{31} - 1)$. Although integers are `object`s, they are immutable. Arithmetic operations that cause overflow lead to undefined behavior in program execution.

2.4.2 Strings

Strings are immutable sequences of characters. String literals are delimited with double qutoes, e.g. `"Hello World"`. Strings support the following three operations: retrieving the length via the `len` function, indexing via the `str[i]` syntax, and concatenation via the `s1 + s2` syntax. Like in Python, ChocoPy does not have a character type. Indexing into a string returns a new string of length 1. Concatenation returns a new string with length equal to the sum of the lengths of its operands.

2.4.3 Lists

Lists are mutable sequences with a fixed length. As such, lists in ChocoPy behave more like arrays in C. A list can be constructed using list literals such as `[1, 2, 3]`. A list of type `T` has the type `[T]`. Like strings, lists support three operations: `len`, indexing via the `lst[i]` syntax, and concatenation via the `l1 + l2` syntax. Indexing a list of type `[T]` returns a value of type `T`. Concatenation of two lists of type `[T]` returns a new list of type `[T]` with the length equal to the sum of the lengths of the two operands. Additionally, lists are mutable and support a fourth operation: element assignment via the syntax `lst[i] = {expr}`.

If `S` is a subtype of `T`, then `[S]` is not a subtype of `[T]`. You cannot assign a value of type `[int]` to a variable of type `[object]`. However, all list types are subtypes of a single common super-type of all lists: `list`.

2.4.4 Objects of user-defined classes

All objects live on the heap and are manipulated using references. That is, `x = cow()` implies that variable `x` references an object of type `cow`. A subsequent assignment `y = x` implies that `x` and `y` reference the *same* `cow` object in memory. Objects are destroyed when they are not reachable from any local, global, or temporary variable.

2.4.5 `None`

`None` is a special value that can be assigned to a variable or attribute of `object` type or any user-defined type.

Notation In the following discussion, we use the following notation: {`expr`} to denote an expression in the program, {`identifier`} to denote an identifier, {`type`} to denote a type annotation, and {`literal`} to denote a constant literal.

2.5 Variable declarations

Local variables in functions can be declared in two ways. First, the formal parameters of a function or method automatically declare local variables. Second, a local variable can be defined using the syntax {`identifier`}:{`type`} `=` {`literal`}. Here, the variable {`identifier`} is declared with type {`type`} and is initialized with a constant literal value such as a boolean, integer, string or `None`. The `global` {`identifier`} statement is used within a function or method body to bind a name to a global variable. Similarly, `nonlocal` {`identifier`} statement is used within a nested function to bind a name to a variable defined in the parent function or body. It is illegal for a `global` declaration to occur at the top-level outside a function body. Similarly, it is illegal for a `nonlocal` declaration to occur outside a nested function. In function and method bodies, all variable declarations—local, global and nonlocal—must appear before any other statement. The same variable may not be declared more than once.

At the top level of the program, global variables can be declared and initialized to a constant using the variable definition syntax: {`identifier`}:{`type`} `=` {`literal`}.

2.6 Expressions

ChocoPy supports the following categories of expressions: literals, identifiers, arithmetic expressions, logical expressions, relational expressions, concatenation expressions, access expressions, and call expressions.

2.6.1 Literals and identifiers

The basic expression is a constant literal or a variable. Literals of type `str`, `bool`, and `int` have been described briefly in Section 2.4, and their lexical structure is described in Section 3.4. Variables evaluate to the value contained in the variable. If an identifier is bound to a global function or class, then it is not a valid expression by itself—it can appear only in specific expressions such as call expressions. This is because ChocoPy does not support first-class functions and classes.

Lists may be constructed in expressions using the list literal syntax `[`{`expr`}`, ...]`. However, this is not considered a constant literal for variable definition statements.

2.6.2 Arithmetic expressions

ChocoPy supports the following binary expressions on operands of type `int`: {`expr`} `+` {`expr`}, {`expr`} `-` {`expr`}, {`expr`} `*` {`expr`}, {`expr`} `//` {`expr`}, and {`expr`} `%` {`expr`}. These operators perform integer addition, subtraction, multiplication, division quotient, and division remainder, respectively. ChocoPy does not support the {`expr`} `/` {`expr`} expression which in Python evaluates to a `float`. The unary expression `-`{`expr`} evaluates to the negative of the integer valued operand. Arithmetic operations return an `int` value.

2.6.3 Logical expressions

ChocoPy supports the following operations on operands of type `bool`: {expr} `and` {expr}, {expr} `or` {expr}, and the unary `not` {expr}, which evaluate to the logical conjunction, disjunction, and negation of their operands, respectively. Logical expressions return a `bool` value.

2.6.4 Relational expressions

ChocoPy supports the following relational expressions on operands of type `int`: {expr} `<` {expr}, {expr} `<=` {expr}, {expr} `>` {expr}, {expr} `>=` {expr}. Additionally, the operands in the expressions of the form {expr} `==` {expr} and {expr} `!=` {expr} can be of types `int, bool, str`. Similarly, the operands in the expressions of the form {expr} `is` {expr} can be `None` or of types other than `int, bool, str`. The `==` and `!=` operators have the following semantics: if both operands are of the same type, which is one of `int`, `bool`, or `str`, then the result is equality or inequality of their values, respectively. The `is` operator can only be applied to `None` and objects of types other than `int`, `str`, and `bool`. {expr} `is` {expr} returns `True` if and only if both operands evaluate to the same object or if both operands evaluate to `None`.

2.6.5 Concatenation expressions

The expression {expr} `+` {expr} can be used to concatenate two strings or two lists; the result is a new string or list, respectively.

2.6.6 Access expressions

An attribute of an object can be accessed using the dot operator: {expr}.{identifier}. For example, `x.y.z` returns the value stored in the attribute `z` of the object obtained by evaluating the expression `x.y`. An element of a string or list can be accessed using the index operator: {expr}[{expr}]. For example, `"Hello"[2+2]` returns the string `"o"`. Accessing a string or list `x` with an index `i` such that `i < 0 or i >= len(x)` aborts the program with an appropriate error message.

2.6.7 Call expressions

A call expression is of the form {identifier}({expr},...), where {expr},... is a comma-separated list of zero or more expressions provided as arguments to the call. If the identifier is bound to a globally declared function, the expression evaluates to the result of the function call. If the identifier is bound to a class, the expression results in the construction of a new object of that class, whose `__init__` method is invoked with the provided arguments.

An expression of the form {expr}.{identifier}({expr},...) invokes a method with name {identifier} and provided list of arguments on the object returned by evaluating the expression to the left of the dot operator. Methods are invoked using dynamic dispatch: if the dynamic type of the object, i.e. the type at the time of execution, is `T`, then the method {identifier} defined in `T` or inherited by `T` is invoked.

2.7 Statements

2.7.1 Control flow

Program statements are executed in order starting from the first top-level statement. The program terminates when the last statement is completely executed. Function invocations transfer control flow to the function's body; once the function completes execution, the program resumes executing the program point just after the function's invocation.

Functions begin execution at their first statement and terminate either due to the execution of a `return` statement or when the last statement in the function body is executed. The absence of a return statement implicitly returns the `None` value. Consider the following example.

```
def bar(x: int) -> object:
    if x > 0:
        return
    elif x == 0:
        return None
    else:
        pass
```

In function `bar`, the execution of the function can terminate either because (1) `x > 0` and a `return` statement with no return value is executed, or (2) `x == 0` and an explicit `return None` is executed, or (3) `x < 0` and the control flow reaches the end of the function, implicitly returning `None`.

2.7.2 Expression statements

The simplest statement is a standalone expression. The expression is evaluated but its result is discarded. These types of statements are useful when they have side-effects, e.g. `print("Hello")`.

2.7.3 Compound statements: conditionals and loops

ChocoPy supports the Python-like `if-elif-else` syntax for conditional control-flow, with `elif`s and `else` being optional. The following code:

```
if {expr1}:
  {body1}
elif {expr2}:
  {body2}
elif {expr3}:
  {body3}
```

is equivalent to:

```
if {expr1}:
  {body1}
else:
  if {expr2}:
    {body2}
  else:
    if {expr3}:
      {body3}
```

ChocoPy supports two types of loops: simple `while` loops and `for` loops over lists and strings. The `for` loops are simply syntactic sugars and can be decomposed into while loops as follows:

```
for x in {expr}:
    {body}
```

is equivalent to:

```
itr = {expr}
idx = 0
while idx < len(itr):
    x = itr[idx]
    {body}
    idx = idx + 1
```

where `itr` and `idx` are fresh variables that are not defined in the original scope. Note that `for` loops do not create new declarations for the loop variables (`x` in the above example); the loop variable must be declared before the `for` statement.

2.7.4 Assignment statements

An assignment statement can be one of the following three forms: (1) {identifier} = {expr}) assigns a value to the variable bound to the identifier, (2) {expr}.{identifier} = {expr}) assigns a value to an attribute of an object, and (3) {expr}[{expr}] = {expr} assigns a value to an element of a list. When assigning a value to index i of a list x, if i < 0 or i >= len(x) then the program aborts after printing an appropriate error message.

Uniquely in assignments, the {expr} on the right-hand side of the = operator can additionally take the form of an *assignment expression*. An assignment expression takes exactly one of the three forms of assignments described in the previous paragraph, and also evaluates to the value of the right-most expression. In this way, assignments can be chained. For example, the code x = y.f = z[0] = 1 assigns the integer value 1 to three memory locations: (1) the first element of the list z, (2) the attribute f of the object referenced by variable y, and (3) the variable x.

2.7.5 Other statements

The pass statement is a no-op. The return statement terminates the execution of a function and optionally returns a value using the return {expr} syntax. It is illegal for a return statement to occur at the top level outside a function or method body.

2.7.6 Predefined classes and functions

The functions print, input and len are provided by the runtime. print is a function that takes a single argument of type str, outputs it to I/O and returns None. The input function takes no arguments and returns a value of type str by reading a line of input from I/O. The function len takes as input one argument x of type object. If x is a str or list, then its length is returned. Otherwise, the program aborts with an error message.

The constructors for object and list take zero arguments each, returning an empty object and an empty list of type object and list, respectively. The constructors for str, int, and bool take one argument x of type object, and produce the following conversions:

- If x is an int, bool(x) returns False if x == 0 and True otherwise.
- If x is a str, bool(x) returns False if len(x) == 0, and True otherwise.
- If x is a bool, str(x) returns the strings "True" and "False" for the corresponding boolean values.
- If x is an int, str(x) returns the string representation of x in base 10.
- If x is a bool, int(x) returns 1 if x == True or 0 if x == False.
- If x is a str, int(x) returns the integer value represented by the string in base 10, in the same way that an integer literal is processed in source code. If x is not a string representation of an integer, then the program aborts with an error message.
- If x is of the same type as being constructed, then x is returned.
- If x is of any other type whose conversion is not listed here, the program aborts with an appropriate error message.

3 Lexical structure

This section describes the details required to implement a lexical analysis for ChocoPy. A lexical analysis reads an input file and produces a sequences of *tokens*. Tokens are matched in the input string using lexical rules that are expressed using regular expressions. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.

The following categories of tokens exist: line structure, identifiers, keywords, literals, operators, and delimiters. Much of this section is borrowed from the lexical analysis of Python[1], but also contains many customizations suited to the simpler syntax of ChocoPy.

3.1 Line structure

In ChocoPy, like in Python, whitespace may be significant both for terminating a statement and for reasoning about the indentation level of a program statement. To accommodate this, ChocoPy defines three lexical tokens that are derived from whitespace: `NEWLINE`, `INDENT`, and `DEDENT`. The rules for when such tokens are generated are described next using the concepts of physical and logical lines.

3.1.1 Physical lines

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files and strings, the following line termination sequences can be used: the Unix form using ASCII LF (`\n`), the Windows form using the sequence ASCII CR LF (`\r\n`), or the old Macintosh form using the ASCII CR (`\r`) character. All of these forms can be used equally, regardless of platform. The end of input also serves as an implicit terminator for the final physical line.

3.1.2 Logical lines

A logical line is a physical line that contains at least one token that is not whitespace or comments. The end of a logical line is represented by the lexical token `NEWLINE`. Statements cannot cross logical line boundaries except where `NEWLINE` is allowed by the syntax (e.g., between statements in control-flow structures such as `while` loops).

3.1.3 Comments

A comment starts with a hash character (`#`) that is not part of a string literal, and ends at the end of the physical line. Comments are ignored by the lexical analyzer; they are not emitted as tokens.

3.1.4 Blank lines

A physical line that contains only spaces, tabs, and possibly a comment, is ignored (i.e., no `NEWLINE` token is generated).

3.1.5 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

Tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line's indentation.

The indentation levels of consecutive lines are used to generate `INDENT` and `DEDENT` tokens, using a stack, as follows: Before the first line of the input program is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one `INDENT` token is generated. If it is smaller, it must be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a `DEDENT` token is generated. At the end of the input program, a `DEDENT` token is generated for each number remaining on the stack that is larger than zero.

[1] `https://docs.python.org/3/reference/lexical_analysis.html`

3.1.6 Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space and tab can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., `ab` is one token, but `a b` is two tokens). Whitespace characters are not tokens; they are simply ignored.

3.2 Identifiers

Identifiers are defined as a contiguous sequence of characters containing the uppercase and lowercase letters `A` through `Z`, the underscore `_` and, except for the first character, the digits `0` through `9`.

3.3 Keywords

The following strings are not recognized as identifiers, and are instead recognized as distinct keyword tokens:

`False`, `None`, `True`, `and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, `yield`.

Not all keywords have special meaning in ChocoPy. For example, ChocoPy does not support `async` or `await`. However, ChocoPy uses the same list of keywords as Python in order to avoid cases where an identifier is legal in ChocoPy but not in Python. Consequently, some keywords (such as `async`) do not appear anywhere in the grammar and will simply lead to a syntax error.

Note that an identifier may contain a keyword as a substring; for example, `classic` is a valid identifier even though it contains the substring `class`. This follows from the longest match rule.

3.4 Literals

String and integer literals are matched at the lexical analysis stage and are represented by string-valued and integer-valued tokens, respectively. The structure of these literals is described below. Boolean literals `True` and `False` are represented simply by their keyword tokens. List literals (e.g. `[1, 2, 3]`) are composed of other tokens, and are therefore recognized in the syntax analysis (ref. Section 4).

3.4.1 String literals

String literals in ChocoPy are greatly simplified from that in Python. In ChocoPy, string literals are simply a sequence of ASCII characters delimited by (and including) double quotes: `"..."`. The ASCII characters must lie within the decimal range 32-126 inclusive—that is, higher than or equal to the *space* character and up to *tilde*. The string itself may contain double quotes escaped by a preceding backslash, e.g. `\"`.

The value of a string token is the sequence of characters between the delimiting double quotes, with any escape sequences applied. The following escape sequences are recognized: `\"`, `\n`, `\t`, `\\` which correspond to a literal double quote, a newline, a tab, and a literal backslash respectively. Any other escape sequence is considered illegal. Some examples follow:

Literal	**Value**
`"Hello"`	`Hello`
`"He\"ll\"o"`	`He"ll"o`
`"He\\\"llo"`	`He\"llo`
`"Hell\o"`	(error: `"\o"` not recognized)

3.4.2 Integer literals

Integer literals in ChocoPy are composed of a sequence of one or more digits `0-9`, where the leftmost digit may only be `0` if it is the only character in the sequence. That is, non-zero valued integer literals may not have leading zeros. The integer value of such literals is interpreted in base 10. The maximum interpreted value can be $2^{31} - 1$ for the literal `2147483647`. A literal with a larger value than this limit results in a lexical error.

3.5 Operators and delimiters

The following is a space-separated list of symbols that correspond to distinct ChocoPy tokens: `+ - * // % < > <= >= == != = ( ) [ ] , : . ->`

4 Syntax

Figure 3 lists the grammar of ChocoPy using an extended BNF notation. The notation ⟦...⟧ is used to group one or more symbols in a production rule and are not tokens in the input language. Symbols or groups may be annotated as follows: '?' denotes that the preceding symbol or group is optional, '$*$' denotes zero or more repeating occurrences and '+' denotes one or more repeating occurrences.

4.1 Precedence

Operators in ChocoPy have the same precedence as that in Python. The following list summarizes the precedence of operators in ChocoPy, from lowest precedence (least binding) to highest precedence (most binding).

- `or`
- `and`
- `not`
- `==`, `!=`, `<`, `>`, `<=`, `>=` , `is`
- `+`, `-` (binary)
- `*`, `//` , `%`
- `-` (unary)
- `.`, `[]`

The arithmetic, logical and dot binary operators are left-associative. The seven comparison operators are non-associative.

5 Type system

This content will be available in a future version of this document.

6 Semantics

This content will be available in a future version of this document.

7 Acknowledgements

ChocoPy is a dialect of Python, version 3.6. The set of Python language features to include in ChocoPy were influenced by Cool (Classroom Object-Oriented Language), which itself is based on Sather164, a dialect of the Sather language. This language manual is largely based off the Cool reference manual.

Several language design choices in ChocoPy were refined through discussions with Rohan Bavishi and Kevin Laeufer. Grant Posner helped review this document and improve its clarity.

$$
\begin{aligned}
\textit{program} &::= [\![\textit{var\_def} \mid \textit{func\_def} \mid \textit{class\_def}]\!]^* \ \textit{stmt}^+ \\
\textit{class\_def} &::= \textbf{class}\ \text{ID}\ \texttt{(}\ \text{ID}\ \texttt{)}\ \texttt{:}\ \text{NEWLINE INDENT}\ [\![\textit{var\_def} \mid \textit{func\_def}]\!]^*\ \text{DEDENT} \\
\textit{func\_def} &::= \textbf{def}\ \text{ID}\ \texttt{(}\ [\![\textit{typed\_var}\ [\![\texttt{,}\ \textit{typed\_var}]\!]^*]\!]^?\ \texttt{)}\ \texttt{->}\ \text{type}\ \texttt{:}\ \text{NEWLINE INDENT}\ \textit{func\_body}\ \text{DEDENT} \\
\textit{func\_body} &::= [\![\textit{global\_decl} \mid \textit{nonlocal\_decl} \mid \textit{var\_def} \mid \textit{func\_def}]\!]^*\ \textit{stmt}^+ \\
\textit{typed\_var} &::= \text{ID}\ \texttt{:}\ \textit{type} \\
\textit{type} &::= \text{ID} \mid \text{STRING} \mid \texttt{[}\ \textit{type}\ \texttt{]} \\
\textit{global\_decl} &::= \textbf{global}\ \text{ID NEWLINE} \\
\textit{nonlocal\_decl} &::= \textbf{nonlocal}\ \text{ID NEWLINE} \\
\textit{var\_def} &::= \textit{typed\_var}\ \texttt{=}\ \textit{literal}\ \text{NEWLINE} \\
\textit{stmt} &::= \textit{simple\_stmt}\ \text{NEWLINE} \\
&\mid \textbf{if}\ \textit{expr}\ \texttt{:}\ \textit{block}\ [\![\textbf{elif}\ \textit{expr}\ \texttt{:}\ \textit{block}\ ]\!]^*\ [\![\textbf{else}\ \texttt{:}\ \textit{block}]\!]^? \\
&\mid \textbf{while}\ \textit{expr}\ \texttt{:}\ \textit{block} \\
&\mid \textbf{for}\ \text{ID}\ \textbf{in}\ \textit{expr}\ \texttt{:}\ \textit{block} \\
\textit{simple\_stmt} &::= \textbf{pass} \\
&\mid \textit{expr} \\
&\mid \textbf{return}\ [\![\textit{expr}]\!]^? \\
&\mid \text{ID}\ \texttt{=}\ \textit{assign\_expr} \\
&\mid \textit{member\_expr}\ \texttt{=}\ \textit{assign\_expr} \\
&\mid \textit{index\_expr}\ \texttt{=}\ \textit{assign\_expr} \\
\textit{block} &::= \text{NEWLINE INDENT}\ \textit{stmt}^+\ \text{DEDENT} \\
\textit{literal} &::= \textbf{None} \\
&\mid \textbf{True} \\
&\mid \textbf{False} \\
&\mid \text{INTEGER} \\
&\mid \text{STRING} \\
\textit{expr} &::= \text{ID} \\
&\mid \textit{literal} \\
&\mid \texttt{[}\ [\![\textit{expr}\ [\![\texttt{,}\ \textit{expr}]\!]^*]\!]^?\ \texttt{]} \\
&\mid \texttt{(}\ \textit{expr}\ \texttt{)} \\
&\mid \textit{member\_expr} \\
&\mid \textit{index\_expr} \\
&\mid \textit{member\_expr}\ \texttt{(}\ [\![\textit{expr}\ [\![\texttt{,}\ \textit{expr}]\!]^*]\!]^?\ \texttt{)} \\
&\mid \text{ID}\ \texttt{(}\ [\![\textit{expr}\ [\![\texttt{,}\ \textit{expr}]\!]^*]\!]^?\ \texttt{)} \\
&\mid \textit{expr}\ \textit{bin\_op}\ \textit{expr} \\
&\mid \textbf{not}\ \textit{expr} \\
&\mid \texttt{-}\ \textit{expr} \\
\textit{bin\_op} &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{//} \mid \texttt{\%} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>} \mid \textbf{and} \mid \textbf{or} \mid \textbf{is} \\
\textit{member\_expr} &::= \textit{expr}\ \texttt{.}\ \text{ID} \\
\textit{index\_expr} &::= \textit{expr}\ \texttt{[}\ \textit{expr}\ \texttt{]} \\
\textit{assign\_expr} &::= \textit{expr} \\
&\mid \text{ID}\ \texttt{=}\ \textit{assign\_expr} \\
&\mid \textit{member\_expr}\ \texttt{=}\ \textit{assign\_expr} \\
&\mid \textit{index\_expr}\ \texttt{=}\ \textit{assign\_expr}
\end{aligned}
$$

Figure 3: Grammar describing the syntax of the ChocoPy language.