# ChocoPy v1.0: Language Manual and Reference

Designed by Rohan Padhye and Koushik Sen

University of California, Berkeley

December 1, 2018

## Contents

## 1 Introduction

This manual describes the ChocoPy language, which is a statically typed dialect 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 has been designed to be a subset of Python. Almost every valid ChocoPy program is also a valid Python 3.6 program. An execution of a ChocoPy program that does not result in error usually has the same observable semantics as the execution of that program in Python 3.6. Appendix A lists the small number of exceptions to this rule.

A ChocoPy program is 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 attribute definitions and method definitions. A class creates a user-defined type. Function definitions can be nested inside other methods and functions. All class names and functions defined at the top level are globally visible. 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.

For readers familiar with the Python language, Figure 1 contains a sample ChocoPy program illustrating top-level functions, statements, global variables, local variables, and type annotations. The type annotations are valid syntaxes in Python 3.6, though the Python interpreter simply ignores these annotations and leaves them as hints for other tools. In contrast, ChocoPy enforces static type checking at compile time. In Figure 1, the function `is_zero` is defined at the top level. Its formal parameters `items` and `idx` are explicitly typed as a list of integers and an integer, respectively. The return type of the function `is_zero` is `bool`. The function defines a local variable, `val`, whose type is `int`. At the top level, the program defines a global variable `mylist`, whose type is a list of integers. Function `is_zero` is invoked in a top level statement and its result is output using the predefined `print` function. Similarly, Figure 2 contains a ChocoPy program that defines two classes: `animal` and `cow`. The class `cow` inherits from `animal`, which in turn inherits from the predefined root class `object`. The Boolean attribute `makes_noise` is defined in `animal` and is therefore inherited by class `cow`. The class `cow` overrides the method `sound`. The constructor for class `cow` is invoked at line 19.

Section 2 provides a detailed but informal overview of the various language constructs in ChocoPy. Sections 3–6 provide formal descriptions of the lexical structure, grammatical syntax, typing rules, and operation semantics of ChocoPy.

## 2 A tour of ChocoPy

**Notation** In the rest of this section, we use:

- {`expr`} to denote an expression in the program.
- {`id`} to denote an identifier such as the name of a variable or function.
- {`stmts`} to denote a list of program statements, separated by newlines.
- {`declarations`} to denote a list of (possibly interleaved) declarations of functions, variables, attributes, and/or classes, where applicable.
- {`type`} to denote a static type annotation.
- {`literal`} to denote a constant literal such as an integer literal, a string literal, or the keywords `True`, `False` or `None`.

```
def is_zero(items: [int], idx: int) -> bool:
    val:int = 0             # Type is explicitly declared
    val = items[idx]
    return val == 0

mylist: [int] = None
mylist = [1, 0, 1]
print(is_zero(mylist, 1))   # Prints 'True'
```

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

## 2.1 The top level

A ChocoPy program consists of zero or more definitions followed by one or more statements, referred to as top-level definitions and statements respectively. Top-level definitions include global variable definitions, function definitions, and class definitions. These definitions create new mappings in a scope called the *global scope*. Global variables are defined using the syntax `{id}:{type} = {literal}`, where the identifier specifies the variable name, the type annotation specifies the static type of the variable, and the constant literal specifies the initial value of the variable upon program execution. The names of global variables, global functions, and classes must be distinct. Top-level statements execute in the global scope; that is, expressions in top-level statements may reference entities defined in the global scope using identifiers. A ChocoPy program's execution begins with the first top-level statement and ends when the last top-level statement is executed completely. Function and class definitions are described in Section 2.2 and Section 2.3 respectively. Program statements are described in Section 2.8.

## 2.2 Functions

In ChocoPy, a function definition can appear at the top level of a program, or it could be nested inside other functions or methods. Functions cannot be redefined in the same scope. However, a function definition in the current scope can shadow a function defined in a surrounding scope of the function. A function definition has the following form:

```
def {id}({id}: {type}, ..., {id}: {type}): -> {type}
  {declarations}
  {stmts}
```

The first line defines the function's name, a comma-separated list of zero or more formal parameters in parentheses, and the function's return type after the '`->`' symbol. Every formal parameter has a name and a static type annotation. The body of a function contains a sequence of zero or more declarations followed by a sequence of one or more program statements. A function definition creates a new scope.

Declarations in a function body include local variable definitions, global and nonlocal variable declarations, and definitions of nested functions. A local variable definition has the form `{id}:{type} = {literal}`. Such a definition declares a local variable with the name `id`, explicitly associates it with a static type, and specifies an initial value using a literal. The `global {id}` statement is used to bind a name to a global variable. Similarly, `nonlocal {id}` statement is used within a nested function to bind a name to a variable defined in the closest surrounding scope which is not the global scope. It is illegal for a `global` declaration to occur at the top level. Similarly, it is illegal for a `nonlocal` declaration to occur outside a nested function. If a variable is not explicitly declared in a function, but is bound to some entity—variable, function, or class—in any surrounding scope, then its binding is implicitly inherited from the surrounding scope as a read-only variable—such a variable cannot be assigned to in any of the function's statements.

## 2.3 Classes

In ChocoPy, a class definition can appear at the top level of a program. Classes cannot be redefined. Class names can never be shadowed; that is, a program may not define any variable or function with the same

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

    def make_noise(self: "animal") -> object:
        if (self.makes_noise):
            print(self.sound())

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

class cow(animal):
    def __init__(self: "cow") -> object:
        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.

name as a class name. A class definition has the following form:

```
class {id}({id}):
  {declarations}
```

The first line specifies the name of the class followed by the name of its *superclass* in parentheses. The class name must not be be bound to any other entity—class, function, or variable—in the program. The superclass must refer to a class that has been previously defined in the program, or be the predefined class `object`. The superclass may not be one of `int`, `str`, or `bool`.

The class body consists of a sequence of attribute definitions and method definitions. In ChocoPy, attributes and methods are associated with object instances of a class, and not with the classes themselves; that is, ChocoPy does not support the notion of *static* class members that some other languages support. Similar to variable definitions, an attribute definition has the form `{id}:{type} = {literal}`. A method definition has the same syntax as a function definitions (ref. Section 2.2), with two important restrictions: (1) a method definition must have at least one formal parameter, and (2) the first formal parameter must have the defining class as type.

A class defines attributes and methods. A class inherits attributes and methods of its superclass. Attributes, whether defined in the current class or inherited from the superclass, cannot be redefined. Methods cannot be redefined in the same class. Inherited methods can be redefined as long as the return type and the types of all formal parameters except the first parameter are exactly the same. Any reference to an attribute or method must be prefixed with an expression and the dot operator (ref. Section 2.6).

If a class `C` is defined to have a superclass `P`, then class `C` is a subclass of `P`. If `C` is a subclass of `P`, then `P` must either be a user-defined class that has been defined before `C` in the program, or be the predefined class `object`. The `object` class does not have a superclass. Since every ChocoPy class (except `object`) inherits attributes and methods from a single superclass, this scheme is called *single inheritance*. The subclass/superclass relation on classes defines a graph. Since a class can only subclass another class defined previously, this graph is a tree with `object` as the root.

In order to create an object `o` of type `C`, the expression `C()` is used. Upon execution, a new object is first created with attributes initialized to their defined values. Then, the `__init__` method in the class `C` is invoked. 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.

## 2.4 Type hierarchy

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 type `C` may be used in place of an object of type `P`, we say that `C` conforms to `P` or that `C` $\leq$ `P` (think: `C` is lower down in the inheritance tree). Conformance of class types is defined in terms of the inheritance graph. Let `A`, `C`, and `P` be types. Then conformance (i.e. $\leq$) is defined as follows:

- `A` $\leq$ `A` for all types `A`
- if `C` is a subclass of `P`, then `C` $\leq$ `P`
- if `A` $\leq$ `C` and `C` $\leq$ `P`, then `A` $\leq$ `P`

The root of the class hierarchy is the predefined class `object`. The predefined types `int`, `bool`, and `str` are subclasses of `object`. 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 are each a list of integers. List types are not related to each other by the $\leq$ relation. Every list type conforms to `object`; that is, `[T]` $\leq$ `object` for any type `T`.

In some situations, we will also need to use the concept of a *join* of two or more types. The join of two types $A$ and $B$ is the least type $C$ (using the $\leq$ ordering) such that $A$ and $B$ conform to $C$. The join operator $\sqcup$ can be formally defined as follows: $C = A \sqcup B$ if and only if:

$$(A \leq C) \wedge (B \leq C) \wedge (\forall D : (A \leq D) \wedge (B \leq D) \Rightarrow (C \leq D))$$

That is, $C$ is the join of $A$ and $B$ if and only if both $A$ and $B$ conform to $C$, and if there is exists type $D$ such that $A$ and $B$ conform to $D$, then $C$ also conforms to $D$. Because the type hierarchy in a ChocoPy program forms a tree, it follows that the join of any two types always exists and is unique; in particular, the join of $A$ and $B$ is the *least common ancestor* of $A$ and $B$ in the type hierarchy.

## 2.5 Values

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

### 2.5.1 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.

### 2.5.2 Booleans

There are exactly two boolean values: `True` and `False`.

### 2.5.3 Strings

Strings are immutable sequences of characters. String literals are delimited with double quotes, e.g. `"Hello World"`. Strings support the following three operations: retrieving the length via the `len` function, indexing via the `s[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.

### 2.5.4 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 the square-brackets notation, e.g. `[1, 2, 3]`.

Like strings, lists of type `[T]` support three operations: `len`, indexing via the `lst[i]` syntax, and concatenation via the `lst1 + lst2` syntax. Indexing a list of type `[T]` returns a value of type `T`. Concatenation of two lists of type $[\mathtt{T}_1]$ and $[\mathtt{T}_2]$ respectively returns a new list of type $[\mathtt{T}_3]$, where the element type of the new list is $\mathtt{T}_3 = \mathtt{T}_1 \sqcup \mathtt{T}_2$. The concatenated list has length equal to the sum of the lengths of the two operands. Additionally, lists of type `[T]` are mutable and support a fourth operation: element assignment via the syntax `lst[i] = {expr}`, where the expression on the right-hand side must conform to the type `T`.

### 2.5.5 Objects of user-defined classes

Objects 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. The `is` operator can be used to determine if two expression reference the same object in memory. Objects are destroyed when they are not reachable from any local, global, or temporary variable.

### 2.5.6 `None`

`None` is a special value that can be assigned to a variable or attribute of type `object`, any user-defined class type, or any list type. The `is` operator can be used to determine if an expression evaluates to the `None` value.

## 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 2.5, 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.

### 2.6.2 List expressions

Lists may be constructed using a comma-separated sequence of expressions delimited by square brackets: `[{expr}, ...]`. The type of a list expression containing one or more elements is `[T]`, where `T` is the *least common ancestor* of the types of the list elements in the program's type hierarchy. In other words, `T` is the least type such that the type of each element expression conforms to `T`. Using the the *join* operator $\sqcup$ defined in Section 2.4, we can say that an expression of the form `[{expr_1}, {expr_2}, ..., {expr_n}]`, where each expression `{expr_i}` has the type `T_i`, results in a list of type `[T]` where `T = T_1` $\sqcup$ `T_2` $\sqcup$ `...` $\sqcup$ `T_n`.

The empty list expression `[]` is handled specially. An empty list expression can appear in places where an expression of a list type is expected. For example, if variable `x` has type `[int]`, then the assignment `x = []` is legal; `x` will contain an empty list of integers after this assignment. In situations other than such special cases, the empty list expression is simply treated as an expression of type `object`.

### 2.6.3 Arithmetic expressions

ChocoPy supports the following arithmetic expressions on two operands each 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` value. The unary expression `-{expr}` evaluates to the negative of the integer-valued operand. Arithmetic operations return an `int` value.

### 2.6.4 Logical expressions

ChocoPy supports the following logical operations on operands of type `bool`: `not {expr}`, `{expr} and {expr}`, and `{expr} or {expr}`, and which evaluate to the logical negation, conjunction, and disjunction of their operands, respectively. Logical expressions return a `bool` value. The binary logical expressions are also *short-circuiting*. If the left operand of an `and` expression evaluates to `False`, then a result of `False` is returned without evaluating the right operand at all. Similarly, if the left operand of an `or` expression evaluates to `True`, then a result of `True` is returned without evaluating the right operand at all. These semantics are important when the expressions in the right-hand side operands contain side-effects.

### 2.6.5 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`, or `str`, as long as both operands are of the same type. In constrast, the operands in the expressions of the form `{expr} is {expr}` can be the `None` literal or expressions of any static type other than `int, bool, str`. The `==` and `!=` operators return true if and only if their operands evaluate to respectively equal or unequal values of integers, booleans, or strings. The `is` operator returns `True` if and only if both operands evaluate to the same object or if both operands evaluate to `None`.

### 2.6.6 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.

### 2.6.7 Access expressions

An attribute of an object can be accessed using the dot operator: `{expr}.{id}`. 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.

### 2.6.8 Call expressions

A call expression is of the form `{id}({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}.{id}({expr},...)` invokes a method with name `{id}` on the object returned by evaluating the expression to the left of the dot operator. The first argument is implicit and is the object whose method is being invoked; the remaining arguments are explicitly provided in parentheses. 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 {id} defined in `T` or inherited by `T` is invoked.

## 2.7 Type annotations

In ChocoPy, static type annotations are used to explicitly provide types for variables, attributes, formal parameters and return types of functions and methods. A type annotation can either refer to a class type `T`, or a list type `[T]` such that `T` is the type annotation corresponding to the type of the elements of the list.

Class-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 to refer to a class type that has not yet been defined, because Python is interpreted line by line. Since we want ChocoPy to behave similarly as Python, we will use the latter form of annotation in the above described scenario. In particular, string literals are always needed in type annotations for the first formal parameter in method definitions, since the type of that parameter is always the same as the enclosing class, which is not yet fully defined.

## 2.8 Statements

### 2.8.1 Expression statements

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

### 2.8.2 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}
```

The expressions in the `if` and `elif` conditions must have type `bool`. The body immediately following an `if` or `elif` condition is only evaluated if the expression evaluates to `True`. If the expression evaluates to `False`, then subsequent `elif` or `else` blocks are considered. The body following the `else` arm is only evaluated if all of the preceding condition expressions evaluate to `False`.

ChocoPy supports two types of loops: simple `while` loops and `for` loops over lists and strings. `while` loops have the following structure:

```
while {expr}:
  {body}
```

The expression must be of type `bool`. The body is repeatedly evaluated as long as the expression evaluates to `True` between iterations.

`for` loops can be used to iterate over elements of a list or characters of a string. They take the following form:

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

for loops are syntactic sugar; the structure above is equivalent to the following de-sugaring:

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

where `len` is the predefined *length* function, and `itr`/`idx` are temporary 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.

### 2.8.3 Assignment statements

An assignment statement can be one of the following three forms: (1) {`id`} `=` {`expr`} assigns a value to the variable bound to the identifier {`id`}, (2) {`expr`}.{`id`} `=` {`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`.

### 2.8.4 Pass statement

The `pass` statement is a no-op. The program state does not change and control flow simply continues on to the next statement.

### 2.8.5 Return statement

The `return` statement terminates the execution of a function and optionally returns a value using the `return` {`expr`} syntax. If a return value is not specified, then the `None` value is returned. It is illegal for a return statement to occur at the top level outside a function or method body. During a function's execution, if control flow reaches the end of the function body without encountering a `return` statement, then the `None` value is implicitly returned. 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`.

In functions or methods that declare a return type of `int`, `str`, or `bool`, all execution paths must contain a `return` statement with an expression that is not a `None` literal. In class definitions, `__init__` methods must not contain any `return` statements.

### 2.8.6 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 `object`, outputs it to I/O and returns the `None` value. 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 a list, then its length is returned. Otherwise, the program aborts with an error message.

The predefined classes `object`, `int`, `bool`, and `str` each define an `__init__` method. The constructors for these classes return an empty object, the value `0`, the value `False`, and the value `""` (empty string) respectively.

## 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.

## 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

The description of indentation is borrowed from the Python 3 documentation[1].

"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."

### 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.

### 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

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

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. Keyword tokens are represented in a **boldfaced font**. Literals and whitespace tokens are represented in UPPERCASE. Nonterminals are formatted *lowercase italics*. Operators and delimiters are formatted as-is. The notation $[\![\ldots]\!]$ 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.

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

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

# 5 Type rules

This section formally defines the type rules of ChocoPy. The type rules define the type of every ChocoPy expression in a given context. The context is the type environment, which describes the type of every unbound identifier appearing in an expression. The type environment is described in Section 5.1. Section 5.2 gives the type rules.

## 5.1 Type environments

To a first approximation, type checking in ChocoPy can be thought of as a bottom-up algorithm: the type of an expression $e$ is inferred from the (previously inferred) types of $e$'s sub-expression. For example, an integer `1` has type `int`; there are no sub-expression in this case. As another example, if the types of $e_1$ and $e_2$ are `int`, then the expression $e_1 > e_2$ has type `bool`.

A complication arises in the case of an expression $v$, where $v$ is a variable or a function. It is not possible to say what the type of $v$ is in a strictly bottom-up algorithm; we need to know the type declared for $v$ in the larger expression. Such a declaration must exist for every variable and function in valid ChocoPy programs.

To capture information about the types of identifiers, we use a *type environment.* The type environment consists of four parts: $O$ a local environment, $M$ a method/attribute environment $M$, $C$ the name of the current class in which the expression or statement appears, and $R$ the return type of the function or method in which the expression or statement appears. $C$ is $\perp$ when the expression or statement appears outside a class, i.e. as a statement or expression in the top level. Similarly, $M$ is $\perp$ when the expression or statement appears outside a function or method, i.e. as a statement or expression in the top level. The local environment and the method/attribute environment are both maps. The local environment is a function of the form:

$$O(v) = T$$

which assigns the type $T$ to a variable $v$. The same environment also holds information about function signatures. For example,

$$O(f) = \{\$ret : T_0, x_1 : T_1, \ldots, x_n : T_n, v_1 : T'_1, \ldots, v_m : T'_m\}$$

gives the "type" of $f$ and denotes that identifier $f$ has formal parameters $x_1, \ldots x_n$ of types $T_1, \ldots, T_n$, respectively, and has return type $T_0$. The identifiers $v_1, \ldots, v_m$ are the variables and nested functions declared in the body of $f$ and their types are $T'_1, \ldots, T'_m$, respectively. The method/attribute environment similarly maps a class and its attributes and methods to their types. For example,

$$M(C, a) = T$$

maps the attribute $a$ in the class $C$ to the type $T$. Similarly,

$$M(C, m) = \{\$ret : T_0, x_1 : T_1, \ldots, x_n : T_n, v_1 : T'_1, \ldots, v_m : T'_m\}$$

maps the method $m$ of class $C$ to it type. Specifically, it denotes that method $m$ in class $C$ has formal parameters $x_1, \ldots x_n$ of types $T_1, \ldots, T_n$, respectively, and has return type $T_0$. The identifiers $v_1, \ldots, v_m$ are the variables and nested functions declared in the body of $m$ and their types are $T'_1, \ldots, T'_m$, respectively.

The third component of the type environment is the name of the class containing the expression or statement to be type checked. The fourth component of the type environment is the return type $R$ of the function or method containing the expression or statement to be type checked.

When type checking function and method definitions, we need to propagate the typing environment from an outer scope to the function scope, where the binding of any identifier is inherited unless the function declares a formal parameter, variable, or a nested function with the same name. Let $O$ be the current local environment and $f$ be a function with type definitions $\{\$ret : T_0, x_1 : T_1, \ldots, x_n : T_n, v_1 : T'_1, \ldots, v_m : T'_m\}$. When type checking the definition of $f$, we type check its body using the local environment $O[T_1/x_1][T_2/x_2] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m]$, where the notation $O[T/c]$ is used to construct a new mapping as follows:

$$O[T/c](c) = T$$
$$O[T/c](d) = O(d) \text{ if } d \neq c$$

## 5.2 Type checking rules

The general form of a type checking rule is:

$$\frac{\vdots}{O, M, C, R \vdash e : T}$$

The rule should be read: in the type environment with local environment $O$, method/attribute environment $M$, containing class $C$, and return type $R$, the expression $e$ has type $T$. The line below the bar is a typing judgment: the turnstyle "$\vdash$" separates context $(O, M, C, R)$ from a proposition $e : T$. The dots above the horizontal bar stand for other judgments about the types of sub-expressions of $e$. These other judgments are hypotheses of the rule; if the hypotheses are satisfied, then the judgment below the bar is true.

The rule for variables is simply that if the environment assigns an idenficier $id$ a type $T$, then the expression $id$ has type $T$.

$$\frac{O(id) = T}{O, M, C, R \vdash id : T} \quad [\textsc{var-read}]$$

The rule for assignment to a variable is more complex. An assignment to a variable can be an expression or a statement. For example, in the statement `x = y = 1`, `y=1` is an expression and `x = e` (where `e` is `y=1`) is a statement.

$$\frac{\begin{array}{l} O(id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq T \end{array}}{O, M, C, R \vdash id = e_1 : T_1} \quad [\textsc{var-assign-expr}]$$

Note that this type rule—as well as others—use the conformance relation $\leq$ (ref. Section 2.4). The rule says that the assigned expression $e_1$ must have a type $T_1$ that conforms to the type $T$ of the identifier $id$ in the type environment. The type of the whole expression is $T_1$.

$$\frac{\begin{array}{l} O(id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq T \end{array}}{O, M, C, R \vdash id = e_1} \quad [\textsc{var-assign-stmt}]$$

The key difference between the rule for variable assignment *expression* and variable assignment *statement* is that the latter does not assign a type to the statement itself, since a statement is not an expression.

Variable and attribute definition are checked in the same manner:

$$\frac{\begin{array}{l} O(id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq T \end{array}}{O, M, C, R \vdash id\colon T = e_1} \quad [\textsc{var-init}]$$

Note that the colon used below the line in the rule for VAR-INIT is the colon in the syntax for type annotations.

A list of definitions and statements type-check if all the component definitions and statements type-check.

$$\frac{\begin{array}{l} O, M, C, R \vdash s_1 \\ O, M, C, R \vdash s_2 \\ \vdots \\ O, M, C, R \vdash s_n \\ n \geq 1 \end{array}}{O, M, C, R \vdash s_1 \text{ NEWLINE } s_2 \text{ NEWLINE } \ldots\ s_n \text{ NEWLINE}} \quad [\text{STMT-DEF-LIST}]$$

The rules for `pass` and expression-statements are trivial:

$$\frac{}{O, M, C, R \vdash \texttt{pass}} \quad [\text{PASS}]$$

$$\frac{O, M, C, R \vdash e : T}{O, M, C, R \vdash e} \quad [\text{EXPR-STMT}]$$

The type rules for constants are also straight-forward:

$$\frac{}{O, M, C, R \vdash \texttt{False} : \mathit{bool}} \quad [\text{BOOL-FALSE}]$$

$$\frac{}{O, M, C, R \vdash \texttt{True} : \mathit{bool}} \quad [\text{BOOL-TRUE}]$$

$$\frac{i \text{ is an integer literal}}{O, M, C, R \vdash i : \mathit{int}} \quad [\text{INT}]$$

$$\frac{s \text{ is a string literal}}{O, M, C, R \vdash s : \mathit{str}} \quad [\text{STR}]$$

The `None` literal is assigned the type `object`:

$$\frac{}{O, M, C, R \vdash \texttt{None} : \mathit{object}} \quad [\text{NONE}]$$

However, several rules handle special cases for assigning `None` to locations that hold values of types other than `int`, `str`, or `bool`. For example:

$$\frac{\begin{array}{l} O(\mathit{id}) = T \\ T \text{ is not one of } \mathit{int}, \mathit{str}, \mathit{bool} \end{array}}{O, M, C, R \vdash \mathit{id} = \texttt{None} : T} \quad [\text{VAR-ASSIGN-EXPR-NONE}]$$

$$\frac{\begin{array}{l} O(\mathit{id}) = T \\ T \text{ is not one of } \mathit{int}, \mathit{str}, \mathit{bool} \end{array}}{O, M, C, R \vdash \mathit{id} = \texttt{None}} \quad [\text{VAR-ASSIGN-STMT-NONE}]$$

$$\frac{\begin{array}{l} O(id) = T \\ T \text{ is not one of } \mathit{int}, \mathit{str}, \mathit{bool} \end{array}}{O, M, C, R \vdash id \colon T = \texttt{None}} \quad [\textsc{var-init-none}]$$

The rules for arithmetic, relational and logical operators are straightforward:

$$\frac{O, M, C, R \vdash e : \mathit{int}}{O, M, C, R \vdash \texttt{-}\ e : \mathit{int}} \quad [\textsc{negate}]$$

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : \mathit{int} \\ O, M, C, R \vdash e_2 : \mathit{int} \\ \mathit{op} \in \{+, -, *, //, \%\} \end{array}}{O, M, C, R \vdash e_1\, \mathit{op}\ e_2 : \mathit{int}} \quad [\textsc{arith}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : \mathit{int} \\ O, M, C, R \vdash e_2 : \mathit{int} \\ \bowtie \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\} \end{array}}{O, M, C, R \vdash e_1 \bowtie e_2 : \mathit{bool}} \quad [\textsc{int-compare}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : \mathit{bool} \\ O, M, C, R \vdash e_2 : \mathit{bool} \\ \bowtie \in \{\texttt{==}, \texttt{!=}\} \end{array}}{O, M, C, R \vdash e_1 \bowtie e_2 : \mathit{bool}} \quad [\textsc{bool-compare}]$$

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : \mathit{bool} \\ O, M, C, R \vdash e_2 : \mathit{bool} \end{array}}{O, M, C, R \vdash e_1 \texttt{ and } e_2 : \mathit{bool}} \quad [\textsc{and}]$$

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : \mathit{bool} \\ O, M, C, R \vdash e_2 : \mathit{bool} \end{array}}{O, M, C, R \vdash e_1 \texttt{ or } e_2 : \mathit{bool}} \quad [\textsc{or}]$$

$$\frac{O, M, C, R \vdash e : \mathit{bool}}{O, M, C, R \vdash \texttt{not}\ e : \mathit{bool}} \quad [\textsc{not}]$$

Strings can be compared, concatenated, and indexed:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : \mathit{str} \\ O, M, C, R \vdash e_2 : \mathit{str} \\ \bowtie \in \{\texttt{==}, \texttt{!=}\} \end{array}}{O, M, C, R \vdash e_1 \bowtie e_2 : \mathit{bool}} \quad [\textsc{str-compare}]$$

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : \mathit{str} \\ O, M, C, R \vdash e_2 : \mathit{str} \end{array}}{O, M, C, R \vdash e_1 \texttt{ + } e_2 : \mathit{str}} \quad [\textsc{str-concat}]$$

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : \mathit{str} \\ O, M, C, R \vdash e_2 : \mathit{int} \end{array}}{O, M, C, R \vdash e_1[e_2] : \mathit{str}} \quad [\textsc{str-select}]$$

The `is` operator can be used to compare values of types other than the special types:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T_1 \\ O, M, C, R \vdash e_2 : T_2 \\ T_1, T_2 \text{ are not one of } \mathit{int}, \mathit{str}, \mathit{bool} \end{array}}{O, M, C, R \vdash e_1 \texttt{ is } e_2 : \mathit{bool}} \quad [\textsc{is}]$$

If $T$ is the name of a class, then object-construction expressions of that class can be typed as follows:

$$\frac{T \text{ is a class}}{O, M, C, R \vdash T() : T} \quad [\textsc{new}]$$

List objects are constructed using a list-literal expression, which results in a list having element type equal to the join of the types of its constituent element expressions:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T_1 \\ O, M, C, R \vdash e_2 : T_2 \\ \vdots \\ O, M, C, R \vdash e_n : T_n \\ n \geq 1 \\ T = \mathit{lub}(T_1, T_2, \ldots, T_n) \end{array}}{O, M, C, R \vdash \texttt{[}e_1, e_2, \ldots, e_n\texttt{]} : [T]} \quad [\textsc{list-literal}]$$

List concatenation results in a list of element type equal to the join of the element types of the operand lists:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T_1] \\ O, M, C, R \vdash e_2 : [T_2] \\ T = \mathit{lub}(T_1, T_2) \end{array}}{O, M, C, R \vdash e_1 \texttt{ + } e_2 : [T]} \quad [\textsc{list-concat}]$$

List elements can be selected, as well as modified using rules similar to variable assignment:

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : [T] \\ O, M, C, R \vdash e_2 : \mathit{int} \end{array}}{O, M, C, R \vdash e_1[e_2] : T} \quad [\textsc{list-select}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T] \\ O, M, C, R \vdash e_2 : \mathit{int} \\ O, M, C, R \vdash e_3 : T_1 \\ T_1 \leq T \end{array}}{O, M, C, R \vdash e_1[e_2] = e_3 : T_1} \quad [\textsc{list-assign-expr}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T] \\ O, M, C, R \vdash e_2 : int \\ O, M, C, R \vdash e_3 : T_1 \\ T_1 \leq T \end{array}}{O, M, C, R \vdash e_1[e_2] = e_3} \quad [\text{LIST-ASSIGN-STMT}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T] \\ O, M, C, R \vdash e_2 : int \\ T \text{ is not one of } int, str, bool \end{array}}{O, M, C, R \vdash e_1[e_2] = \texttt{None} : T} \quad [\text{LIST-ASSIGN-EXPR-NONE}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : [T] \\ O, M, C, R \vdash e_2 : int \\ T \text{ is not one of } int, str, bool \end{array}}{O, M, C, R \vdash e_1[e_2] = \texttt{None}} \quad [\text{LIST-ASSIGN-STMT-NONE}]$$

A special case of a list literal is an empty list, which is assigned type `object`:

$$\frac{}{O, M, C, R \vdash \texttt{[]} : object} \quad [\text{NIL}]$$

Empty lists may be assigned to variables, list elements, and other places where a list type is expected:

$$\frac{O(id) = [T]}{O, M, C, R \vdash id = \texttt{[]} : [T]} \quad [\text{VAR-ASSIGN-EXPR-NIL}]$$

$$\frac{O(id) = [T]}{O, M, C, R \vdash id = \texttt{[]}} \quad [\text{VAR-ASSIGN-STMT-NIL}]$$

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : [[T]] \\ O, M, C, R \vdash e_2 : int \end{array}}{O, M, C, R \vdash e_1[e_2] = \texttt{[]} : [T]} \quad [\text{LIST-ASSIGN-EXPR-NIL}]$$

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : [[T]] \\ O, M, C, R \vdash e_2 : int \end{array}}{O, M, C, R \vdash e_1[e_2] = \texttt{[]}} \quad [\text{LIST-ASSIGN-STMT-NIL}]$$

In order to type check attribute accesses, we look up the class-member environment $M$:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ M(T_0, id) = T \end{array}}{O, M, C, R \vdash e_0.id : T} \quad [\text{ATTR-READ}]$$

$$\frac{\begin{array}{l} M(C, id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq T \end{array}}{O, M, C, R \vdash id\colon T = e_1} \quad [\text{ATTR-INIT}]$$

$$\frac{\begin{array}{l} M(C, id) = T \\ T \text{ is not one of } int, str, bool \end{array}}{O, M, C, R \vdash id\colon T = \texttt{None}} \quad [\text{ATTR-INIT-NONE}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ O, M, C, R \vdash e_1 : T_1 \\ M(T_0, id) = T \\ T_1 \leq T \end{array}}{O, M, C, R \vdash e_0.id = e_1 : T_1} \quad [\text{ATTR-ASSIGN-EXPR}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ M(T_0, id) = T \\ T \text{ is not one of } int, str, bool \end{array}}{O, M, C, R \vdash e_0.id = \texttt{None} : T} \quad [\text{ATTR-ASSIGN-EXPR-NONE}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ M(T_0, id) = [T] \end{array}}{O, M, C, R \vdash e_0.id = \texttt{[]} : [T]} \quad [\text{ATTR-ASSIGN-EXPR-NIL}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ O, M, C, R \vdash e_1 : T_1 \\ M(T_0, id) = T \\ T_1 \leq T \end{array}}{O, M, C, R \vdash e_0.id = e_1} \quad [\text{ATTR-ASSIGN-STMT}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ M(T_0, id) = T \\ T \text{ is not one of } int, str, bool \end{array}}{O, M, C, R \vdash e_0.id = \texttt{None}} \quad [\text{ATTR-ASSIGN-STMT-NONE}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : T_0 \\ M(T_0, id) = [T] \end{array}}{O, M, C, R \vdash e_0.id = \texttt{[]}} \quad [\text{ATTR-ASSIGN-STMT-NIL}]$$

The rule to type check a function-invocation expression is more complex:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T_1'' \\ O, M, C, R \vdash e_2 : T_2'' \\ \vdots \\ O, M, C, R \vdash e_n : T_n'' \\ n \geq 0 \\ O(f) = \{\$ret : T_0, x_1 : T_1, \ldots, x_n : T_n, v_1 : T_1', \ldots, v_m : T_m'\} \\ \forall 1 \leq i \leq n : (T_i'' \leq T_i \text{ or } (e_i \text{ is } \texttt{None} \text{ and } T_i \text{ is not one of } \mathit{int}, \mathit{str}, \mathit{bool}) \text{ or } (e_i \text{ is } \texttt{[]} \text{ and } T_i \text{ is a list type})) \end{array}}{O, M, C, R \vdash f(e_1, e_2, \ldots, e_n) : T_0} \quad [\text{INVOKE}]$$

To type check a function invocation, each of the arguments to the function must be first type checked. The type of each argument must conform to the types of the corresponding formal parameter of the function, except if the argument is `None` or `[]` which are handled separately. The invocation expression is assigned the function's declared return type.

Method dispatch expressions are type checked in a similar fashion. The key difference from the function invocation expression is that the target method is determined by consulting the method/attribute environment using the type of the receiver object expression:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_1 : T_1'' \\ O, M, C, R \vdash e_2 : T_2'' \\ \vdots \\ O, M, C, R \vdash e_n : T_n'' \\ n \geq 1 \\ M(T_1'', f) = \{\$ret : T_0, x_1 : T_1, \ldots, x_n : T_n, v_1 : T_1', \ldots, v_m : T_m'\} \\ T_1'' \leq T_1 \\ \forall 1 \leq 2 \leq n : (T_i'' \leq T_i \text{ or } (e_i \text{ is } \texttt{None} \text{ and } T_i \text{ is not one of } \mathit{int}, \mathit{str}, \mathit{bool}) \text{ or } (e_i \text{ is } \texttt{[]} \text{ and } T_i \text{ is a list type})) \end{array}}{O, M, C, R \vdash e_1.f(e_2, \ldots, e_n) : T_0} \quad [\text{DISPATCH}]$$

Return statements are type checked using the return-type environment:

$$\frac{\begin{array}{l} O, M, C, R \vdash e : T \\ T \leq R \end{array}}{O, M, C, R \vdash \texttt{return } e} \quad [\text{RETURN-E}]$$

$$\frac{R \text{ is not one of } \mathit{int}, \mathit{str}, \mathit{bool}}{O, M, C, R \vdash \texttt{return None}} \quad [\text{RETURN-NONE}]$$

$$\frac{R \text{ is not one of } \mathit{int}, \mathit{str}, \mathit{bool}}{O, M, C, R \vdash \texttt{return}} \quad [\text{RETURN}]$$

$$\frac{R = [T]}{O, M, C, R \vdash \texttt{return []}} \quad [\text{RETURN-NIL}]$$

Type rules for other statements are as follows:

$$\frac{\begin{array}{l} O, M, C, R \vdash e_0 : bool \\ O, M, C, R \vdash b_0 \\ O, M, C, R \vdash e_1 : bool \\ O, M, C, R \vdash b_1 \\ \vdots \\ O, M, C, R \vdash e_n : bool \\ O, M, C, R \vdash b_n \\ n \geq 0 \\ O, M, C, R \vdash b_{n+1} \end{array}}{O, M, C, R \vdash \texttt{if}\ e_0 \texttt{:} b_0\ \texttt{elif}\ e_1 \texttt{:} b_1 \ldots\ \texttt{elif}\ e_n \texttt{:} b_n\ \texttt{else:} b_{n+1}} \quad [\textsc{if-elif-else}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e : bool \\ O, M, C, R \vdash b \end{array}}{O, M, C, R \vdash \texttt{while}\ e \texttt{:} b} \quad [\textsc{while}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e : str \\ O(id) = T \\ str \leq T \\ O, M, C, R \vdash b \end{array}}{O, M, C, R \vdash \texttt{for}\ id\ \texttt{in}\ e \texttt{:} b} \quad [\textsc{for-str}]$$

$$\frac{\begin{array}{l} O, M, C, R \vdash e : [T_1] \\ O(id) = T \\ T_1 \leq T \\ O, M, C, R \vdash b \end{array}}{O, M, C, R \vdash \texttt{for}\ id\ \texttt{in}\ e \texttt{:} b} \quad [\textsc{for-list}]$$

The rule for typing a function definition for $f$ checks the body of the function $f$ in an environment where $O$ is extended with bindings for the names explicitly declared by $f$.

$$\frac{\begin{array}{l} O(f) = \{\$ret : T_0, x_1 : T_1, \ldots, x_n : T_n, v_1 : T'_1, \ldots, v_m : T'_m\} \\ n \geq 0 \qquad m \geq 0 \\ O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m], M, C, T_0 \vdash b \end{array}}{O, M, C, R \vdash \texttt{def}\ f(x_1 \texttt{:} T_1, \ldots, x_n \texttt{:} T_n)\ \texttt{->}\ T_0 \texttt{:} b} \quad [\textsc{func-def}]$$

$$\frac{\begin{array}{l} O(C, f) = \{\$ret : T_0, x_1 : T_1, \ldots, x_n : T_n, v_1 : T'_1, \ldots, v_m : T'_m\} \\ n \geq 1 \qquad m \geq 0 \\ C = T_1 \\ O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m], M, C, T_0 \vdash b \end{array}}{O, M, C, R \vdash \texttt{def}\ f(x_1 \texttt{:} T_1, \ldots, x_n \texttt{:} T_n)\ \texttt{->}\ T_0 \texttt{:} b} \quad [\textsc{method-def}]$$

Class definitions are type checked by propagating the appropriate typing environment:

$$\frac{O, M, C, R \vdash b}{O, M, \bot, R \vdash \texttt{class}\ C(S) \texttt{:} b} \quad [\textsc{class-def}]$$

The global typing environment always contains at least the following predefined functions and class methods:

$$\begin{aligned}
O(len) &= \{\$ret : int,\ arg : object\} \\
O(print) &= \{\$ret : object,\ arg : object\} \\
O(input) &= \{\$ret : str\} \\
M(object, \_\_init\_\_) &= \{\$ret : object,\ self : object\} \\
M(str, \_\_init\_\_) &= \{\$ret : object,\ self : object\} \\
M(int, \_\_init\_\_) &= \{\$ret : object,\ self : object\} \\
M(bool, \_\_init\_\_) &= \{\$ret : object,\ self : object\}
\end{aligned}$$

## 6 Operational semantics

This section contains the formal operational semantics for the ChocoPy language. The operational semantics define how every definition, statement, or expression in a ChocoPy program should be evaluated in a given context. The context has four components: a global environment, a local environment, a store, and a return value. Section 6.1 describes these components. Section 6.2 defines the syntax used to refer to ChocoPy values, and Section 6.3 defines the syntax used to refer to class definitions.

Keep in mind that a formal semantics is a specification only—it does not describe an implementation. The purpose of presenting the formal semantics is to make clear all the details of the behavior of a ChocoPy program. How this behavior is implemented is another matter.

### 6.1 Evaluation context

The value of a ChocoPy expression depends on the context in which it is evaluated. The context comprises of an *environment*, which maps variable identifiers to *locations*. Intuitively, an environment tells us for a given identifier the address of the memory location where that identifier's value is stored. For a given expression, the environment must assign a location to all identifiers to which the expression may refer. For the expression $a + b$, we need an environment that maps $a$ to some location and $b$ to some location. We'll use the following syntax to describe environments, which is very similar to the syntax of type environments defined in Section 5.1.

$$E = [a : l_1, b : l_2]$$

This environment maps variable $a$ to location $l_1$ and variable $b$ to location $l_2$.

The second component of the evaluation context is the *store* (memory). The store maps locations to values, where values in ChocoPy are objects or functions. Intuitively, a store tells us what value is stored in a given memory location. For the moment, assume all values are integers. A store is similar to an environment:

$$S = [l_1 \mapsto 42, l_2 \mapsto 7]$$

This store maps location $l_1$ to the value 42 and the location $l_2$ to the value 7. Given an environment and a store, the value of a variable can be retrieved by first looking up the location of a variable in the environment $E$, and then looking up the value stored at this location in the store $S$. For example, the value of variable $a$ can be looked up in the following way:

$$\begin{aligned}
E(a) &= l_1 \\
S(l_1) &= 42
\end{aligned}$$

Together, the environment and the store define the execution state at a particular step of the evaluation of a ChocoPy program. The double indirection from identifiers to locations to values allows us to model variables. Consider what happens if the value 11 is assigned to variable $a$ in the environment and store defined above. Assigning to a variable means changing the value to which it refers but not its location. To perform the assignment, we look up the location for $a$ in the environment E and then change the mapping for the obtained location to the new value, giving a new store $S'$.

$$\begin{aligned} E(a) &= l_1 \\ S' &= S[11/l_1] \end{aligned}$$

Where the syntax $S' = S[v/l]$ denotes a new store $S'$ that is identical to the store $S$, except that $S'$ maps location $l$ to value $v$. It is formally defined as follows:

$$\begin{aligned} S[v/l](l) &= v \\ S[v/l](l') &= S(l') \text{ if } l' \neq l \end{aligned}$$

There are also situations in which the environment is modified. Consider the definition of a variable in ChocoPy:

$$\texttt{x} : \texttt{int} = \texttt{36}$$

When evaluating this definition, we must introduce a new identifier $x$ into the environment, which will be valid for the rest of the scope in which definition occurs. If the current environment and store are $E$ and $S$ respectively, then the new environment and store after evaluating this definition are $E'$ and $S'$ respectively, which are defined by:

$$\begin{aligned} l_x &= newloc(S) \\ E' &= E[l_x/x] \\ S' &= S[36/l_x] \end{aligned}$$

The first step is to allocate a location for the variable $x$. The location should be fresh, meaning that the current store should not have a mapping for it. The function *newloc* applied to a store gives us an unused location in that store. We then create a new environment $E'$, which maps $x$ to $l_x$ but also contains all of the mappings of $E$ for identifiers other than $x$. Note that if $x$ already has a mapping in E, the new environment $E'$ hides this old mapping. We must also update the store to map the new location to a value. In this case $l_x$ maps to the value 36, which is the initial value for $x$ specified in the variable's definition.

The example in this subsection oversimplifies ChocoPy environments and stores a bit, because simple integers are not ChocoPy values. Even integers are full-fledged objects in ChocoPy.

In ChocoPy, the evaluation context consists of two additional components: a global environment $G$, and a return value $R$. The global environment $G$ is an environment similar to $E$, but it always contains mappings for variables and functions defined at the global scope. This environment is useful for handling cases where a nested function references a global variable `x` via the `global x` declaration, bypassing any inherited mappings for variable `x` from the enclosing function. The context $R$ is used within function bodies to keep track of whether a `return` statement has been executed; this is important since the execution of a `return` statement terminates the execution of a function body. If a `return` statement has been executed in the current function, then $R$ represents the returned value; otherwise, we use the '_' (underscore) symbol to denote the absence of any returned value.

## 6.2 Syntax for values

To describe ChocoPy semantics, we use the following categories of values: objects that are instances of classes, list objects, the `None` value, and functions (or methods).

### 6.2.1 Class instances

Let $v$ be a value corresponding to a object belonging to class $X$. The value $v$ is represented by the syntax:

$$v = X(a_1 = l_1, a_2 = l_2, \ldots, a_n = l_n)$$

where each $a_i$ for $1 \leq i \leq n$ is an attribute or method (including those inherited) of class $X$, and each $l_i$ is the location where the value of attribute or method $a_i$ is stored. Note that each $a_i$ is distinct in a semantically valid ChocoPy program.

For base objects of ChocoPy (i.e., `int`, `str`, `bool`), we use a special case of the above syntax. Base objects have a class name, but their attributes are not like attributes of normal classes, because they cannot be modified. Therefore, we describe base objects using the following syntax:

$$\begin{aligned} int(5) \\ bool(True) \\ str(7, \text{“ChocoPy”}) \end{aligned}$$

Integers and booleans contain just one component: the integer or boolean value respectively. Strings contain two components: a length and the actual sequence of ASCII characters.

### 6.2.2 List objects

A list object $v$ of length $n$ is represented by the syntax:

$$v = [l_1, l_2, \ldots, l_n]$$

where each $l_i$ is the location of the element at index $i - 1$.

### 6.2.3 `None`

The special value `None` is represented simply as *None*.

### 6.2.4 Functions

Functions are not first-class values in ChocoPy; that is, ChocoPy variables cannot store references to functions and ChocoPy expressions cannot evaluate to a function. However, functions are represented as values in the formal semantics and are the result of evaluating function and method definitions. Global functions, nested functions, and methods of classes are all represented by the same syntax, which is as follows:

$$v = (x_1, \ldots, x_n, y_1 = e_1, \ldots, y_k = e_k, b_{body}, E_f)$$

Here, $v$ is a function having $n$ formal parameters $x_1, \ldots, x_n$ and $k$ local definitions $y_1, \ldots, y_k$. Local definitions include local variables and nested functions. The term $e_i$ represents the definition corresponding to the identifier $y_i$. For variable definitions, $e_i$ is the literal expression which gives its initial value. For nested functions, $e_i$ is the function definition itself. The term $b_{body}$ is the function's body, which is a sequence of statements. The term $E_f$ is the environment in which the function is defined, with appropriate substitutions for `global` declarations within the function.

## 6.3 Syntax for class definitions

When referring to class definitions, we need to use an additional notation. The mapping *class* is used to get the attributes and methods defined in a particular class. The syntax is as follows:

$$class(A) = (a_1 = e_1, \ldots, a_m = e_m)$$

where each $a_i$ is an attribute or method belonging to the class $A$ (including inherited members). If $a_i$ is an attribute, then the corresponding $e_i$ is the literal expression that specifies the attribute's initial value. If $a_i$ is a method name, then $e_i$ is the corresponding method definition (which has the same syntax as a function definition).

For classes that inherit the method `__init__` from its definition in class `object`, we assume that the method body contains a single statement: `pass`.

## 6.4 Operational rules

Equipped with the notation for environments, stores, return context, and the syntax for values and class definitions, we can now present the formal operational semantics rules for ChocoPy. The general form of a rule is:

$$\frac{\vdots}{G, E, S, R \vdash e : v, S', R'}$$

This rule should be read as follows: in a context with global environment $G$, local environment $E$, store $S$, and return value $R$, the program fragment $e$ evaluates to value $v$, after which the new store is $S'$ and the returned value is $R'$. Program fragments include expressions, definitions, statements, and sequences of statements. The value $v$ will only be meaningful when $e$ is an expression or function/method definition; in other cases, we represent the value $v$ by the '_' (underscore) symbol to denote the absence of any value.

We begin with the trivial rules for constant literals and the `pass` statement:

$$\frac{}{G, E, S, \_ \vdash \texttt{None} : \mathit{None}, S, \_} \quad [\textsc{none}]$$

$$\frac{}{G, E, S, \_ \vdash \texttt{False} : \mathit{bool}(\mathit{false}), S, \_} \quad [\textsc{bool-false}]$$

$$\frac{}{G, E, S, \_ \vdash \texttt{True} : \mathit{bool}(\mathit{true}), S, \_} \quad [\textsc{bool-true}]$$

$$\frac{i \text{ is an integer literal}}{G, E, S, \_ \vdash i : \mathit{int}(i), S, \_} \quad [\textsc{int}]$$

$$\frac{\begin{array}{l} s \text{ is a string literal} \\ n \text{ is the length of the string } s \end{array}}{G, E, S, \_ \vdash s : \mathit{str}(n, s), S, \_} \quad [\textsc{str}]$$

$$\frac{}{G, E, S, \_ \vdash \texttt{pass} : \_, S, \_} \quad [\textsc{pass}]$$

An expression-statement evaluates an expression, which may modify the store, and then discards its value:

$$\frac{G, E, S, \_ \vdash e : v, S', \_}{G, E, S, \_ \vdash e : \_, S', \_} \quad [\textsc{expr-stmt}]$$

The rules for accessing variables require look-ups in the environment and store. The rules for variable assignment also modify the store.

$$\frac{\begin{array}{l} E(id) = l_{id} \\ S(l_{id}) = v \end{array}}{G, E, S, \_ \vdash id : v, S, \_} \quad [\text{VAR-READ}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : v, S_1, \_ \\ E(id) = l_{id} \\ S_2 = S_1[v/l_{id}] \end{array}}{G, E, S, \_ \vdash id = e : v, S_2, \_} \quad [\text{VAR-ASSIGN-EXPR}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : v, S_1, \_ \\ E(id) = l_{id} \\ S_2 = S_1[v/l_{id}] \end{array}}{G, E, S, \_ \vdash id = e : \_, S_2, \_} \quad [\text{VAR-ASSIGN-STMT}]$$

The last two rules should be read as follows: the right-hand side of the assignment is first evaluated with the context $G, E, S, \_$ to produce value $v$; this evaluation may result in a modified store $S_1$. After evaluating the variable assignment, the new store is $S_2$, which is a modification of $S_1$ where the location $l_{id}$ maps to the evaluated value of the right-hand-side: $v$. The only difference between a variable assignment expression and a variable assignment statement is that the former evaluates to the same value as the right-hand side expression, while the latter does not evaluate to a value.

The rules for arithmetic and relational operators are defined in terms of the mathematical evaluation of the corresponding operators.

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : int(i_1), S_1, \_ \\ v = int(-i_1) \end{array}}{G, E, S, \_ \vdash \texttt{-}\ e : v, S_1, \_} \quad [\text{NEGATE}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_1 : int(i_1), S_1, \_ \\ G, E, S, \_ \vdash e_2 : int(i_2), S_1, \_ \\ op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{//}, \texttt{\%}\} \\ op \in \{\texttt{//}, \texttt{\%}\} \Rightarrow i_2 \neq 0 \\ v = int(i_1\ op\ i_2) \end{array}}{G, E, S, \_ \vdash e_1\ op\ e_2 : v, S_2, \_} \quad [\text{ARITH}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_1 : int(i_1), S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : int(i_2), S_2, \_ \\ \bowtie \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\} \\ v = \begin{cases} bool(true) & \text{if } i_1 \bowtie i_2 \\ bool(false) & \text{otherwise} \end{cases} \end{array}}{G, E, S, \_ \vdash e_1 \bowtie e_2 : v, S_2, \_} \quad [\text{INT-COMPARE}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_1 : \mathit{bool}(b_1), S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : \mathit{bool}(b_2), S_2, \_ \\ \bowtie \in \{\texttt{==}, \texttt{!=}\} \\ v = \begin{cases} \mathit{bool}(\mathit{true}) & \text{if } b_1 \bowtie b_2 \\ \mathit{bool}(\mathit{false}) & \text{otherwise} \end{cases} \end{array}}{G, E, S, \_ \vdash e_1 \bowtie e_2 : v, S_2, \_} \quad [\textsc{bool-compare}]$$

String operations are defined analogusly:

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_1 : \mathit{str}(n_1, s_1), S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : \mathit{str}(n_2, s_2), S_2, \_ \\ \bowtie \in \{\texttt{==}, \texttt{!=}\} \\ v = \begin{cases} \mathit{bool}(\mathit{true}) & \text{if } \bowtie \text{ is } \texttt{=} \text{ and } s_1 = s_2 \\ \mathit{bool}(\mathit{true}) & \text{if } \bowtie \text{ is } \texttt{!=} \text{ and } s_1 \neq s_2 \\ \mathit{bool}(\mathit{false}) & \text{otherwise} \end{cases} \end{array}}{G, E, S, \_ \vdash e_1 \bowtie e_2 : v, S_2, \_} \quad [\textsc{str-compare}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_1 : \mathit{str}(n_1, s_1), S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : \mathit{str}(n_2, s_2), S_2, \_ \\ v = \mathit{str}(n_1 + n_2, s_1.s_2) \qquad \text{where } s_1.s_2 \text{ is the concatenated string} \end{array}}{G, E, S, \_ \vdash e_1 + e_2 : v, S_2, \_} \quad [\textsc{str-concat}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_1 : \mathit{str}(n, c_1.c_2 \ldots c_n), S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : \mathit{int}(i), S_2, \_ \\ 0 \leq i < n \\ v = \mathit{str}(1, c_{i+1}) \end{array}}{G, E, S, \_ \vdash e_1[e_2] : v, S_2, \_} \quad [\textsc{str-select}]$$

The `is` operator tests whether its operands reference the same object in memory:

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_1 : v_1, S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : v_2, S_2, \_ \\ v = \begin{cases} \mathit{bool}(\mathit{true}) & \text{if } v_1 = \mathit{None} \text{ and } v_2 = \mathit{None} \\ \mathit{bool}(\mathit{true}) & \text{if } v_1 \text{ and } v_2 \text{ are the same object in memory} \\ \mathit{bool}(\mathit{false}) & \text{otherwise} \end{cases} \end{array}}{G, E, S, \_ \vdash e_1 \texttt{ is } e_2 : v, S_2, \_} \quad [\textsc{is}]$$

The rule for the unary logical operator `not` is simple: the operand's value is negated. The rules for the binary logical operators perform short-circuit evaluation: if the result of logical conjunction or disjunction is apparent from evaluating the first operand (because the operand corresponds to `False` or `True` respectively), then the second operand is not evaluated at all. Otherwise, the result of the conjunction or disjunction is equal to the value of the second operand.

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : \mathit{bool}(b), S_1, \_ \\ v = \begin{cases} \mathit{bool}(\mathit{false}) & \text{if } b = \mathit{true} \\ \mathit{bool}(\mathit{true}) & \text{otherwise} \end{cases} \end{array}}{G, E, S, \_ \vdash \texttt{not } e : v, S_1, \_} \quad [\textsc{not}]$$

$$\frac{G, E, S, \_ \vdash e_1 : \mathit{bool}(\mathit{false}), S_1, \_}{G, E, S, \_ \vdash e_1 \texttt{ and } e_2 : \mathit{bool}(\mathit{false}), S_1, \_} \quad [\text{AND-1}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_1 : \mathit{bool}(\mathit{true}), S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : v, S_2, \_ \end{array}}{G, E, S, \_ \vdash e_1 \texttt{ and } e_2 : v, S_2, \_} \quad [\text{AND-2}]$$

$$\frac{G, E, S, \_ \vdash e_1 : \mathit{bool}(\mathit{true}), S_1, \_}{G, E, S, \_ \vdash e_1 \texttt{ or } e_2 : \mathit{bool}(\mathit{true}), S_1, \_} \quad [\text{OR-1}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_1 : \mathit{bool}(\mathit{false}), S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : v, S_2, \_ \end{array}}{G, E, S, \_ \vdash e_1 \texttt{ or } e_2 : v, S_2, \_} \quad [\text{OR-2}]$$

`if-else`, and `if-elif-else` statements are evaluated by first evaluating the condition, and then deciding which branch to evaluate. Note that if the body of the conditional branch may return a value, which is propagated by the `if` statement as well.

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : \mathit{bool}(\mathit{true}), S_1, \_ \\ G, E, S_1, \_ \vdash b_1 : \_, S_2, R \end{array}}{G, E, S, \_ \vdash \texttt{if } e\texttt{: } b_1 \texttt{ else: } b_2 : \_, S_2, R} \quad [\text{IF-ELSE-TRUE}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : \mathit{bool}(\mathit{false}), S_1, \_ \\ G, E, S_1, \_ \vdash b_2 : \_, S_2, R \end{array}}{G, E, S, \_ \vdash \texttt{if } e\texttt{: } b_1 \texttt{ else: } b_2 : \_, S_2, R} \quad [\text{IF-ELSE-FALSE}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : \mathit{bool}(\mathit{true}), S_1, \_ \\ G, E, S_1, \_ \vdash b_1 : \_, S_2, R \end{array}}{G, E, S, \_ \vdash \texttt{if } e_0\texttt{:}b_0 \texttt{ elif } e_1\texttt{:}b_1 \ldots \texttt{ elif } e_n\texttt{:}b_n \texttt{ else:}b_{n+1} : \_, S_2, R} \quad [\text{IF-ELIF-TRUE}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_0 : \mathit{bool}(\mathit{false}), S_1, \_ \\ G, E, S_1, \_ \vdash \texttt{if } e_1\texttt{:}b_1 \ldots \texttt{ elif } e_n\texttt{:}b_n \texttt{ else:}b_{n+1} : \_, S_2, R \end{array}}{G, E, S, \_ \vdash \texttt{if } e_0\texttt{:}b_0 \texttt{ elif } e_1\texttt{:}b_1 \ldots \texttt{ elif } e_n\texttt{:}b_n \texttt{ else:}b_{n+1} : \_, S_2, R} \quad [\text{IF-ELIF-FALSE}]$$

$$\frac{G, E, S, \_ \vdash \texttt{if } e\texttt{: } b_1 \texttt{ else: pass} : \_, S_1, R}{G, E, S, \_ \vdash \texttt{if } e\texttt{: } b_1 : \_, S_1, R} \quad [\text{IF-NO-ELSE}]$$

`while` loops are evaluated by first evaluating the condition. If the condition is $false$, the loop terminates. If the condition is $true$, then the loop body is executed; the `while` loop is then evaluated again unless the loop body returns a value.

$$\frac{G, E, S, \_ \vdash e : bool(false), S_1, \_}{G, E, S, \_ \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_1, \_} \quad [\text{WHILE-FALSE}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : bool(true), S_1, \_ \\ G, E, S_1, \_ \vdash b : \_, S_2, \_ \\ G, E, S_2, \_ \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_3, R \end{array}}{G, E, S, \_ \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_3, R} \quad [\text{WHILE-TRUE-LOOP}]$$

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : bool(true), S_1, \_ \\ G, E, S_1, \_ \vdash b : \_, S_2, R \\ R \text{ is not } \_ \end{array}}{G, E, S, \_ \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_2, R} \quad [\text{WHILE-TRUE-RETURN}]$$

The `return` statement explicitly or implicitly sets the return value $R$.

$$\frac{G, E, S, \_ \vdash e : v, S_1, \_}{G, E, S, \_ \vdash \texttt{return}\ e : \_, S_1, v} \quad [\text{RETURN-E}]$$

$$\frac{}{G, E, S, \_ \vdash \texttt{return} : \_, S_1, None} \quad [\text{RETURN}]$$

A sequence of statements is evaluated by evaluating each statement in the sequence in turn, either until some statement returns a value or until the last statement in the sequence is evaluated.

$$\frac{\begin{array}{l} n \geq 0 \\ G, E, S_0, \_ \vdash s_1 : \_, S_1, \_ \\ G, E, S_1, \_ \vdash s_2 : \_, S_2, \_ \\ \vdots \\ G, E, S_{n-1}, \_ \vdash s_n : \_, S_n, \_ \end{array}}{G, E, S_0, \_ \vdash s_1 \text{ NEWLINE } s_2 \text{ NEWLINE } \ldots\ s_n \text{ NEWLINE } : \_, S_n, \_} \quad [\text{STMT-SEQ}]$$

$$\frac{\begin{array}{l} n \geq 0 \\ G, E, S_0, \_ \vdash s_1 : \_, S_1, \_ \\ G, E, S_1, \_ \vdash s_2 : \_, S_2, \_ \\ \vdots \\ G, E, S_{k-1}, \_ \vdash s_k : \_, S_k, R \\ k \leq n, \qquad R \text{ is not } \_ \end{array}}{G, E, S_0, \_ \vdash s_1 \text{ NEWLINE } s_2 \text{ NEWLINE } \ldots\ s_n \text{ NEWLINE } : \_, S_k, v} \quad [\text{STMT-SEQ-RETURN}]$$

The rule for evaluating a function invocation expression is complex:

$$\frac{\begin{array}{l} S_0(E(f)) = (x_1, \ldots, x_n, y_1 = e'_1, \ldots, y_k = e'_k, b_{body}, E_f) \\ n, k \geq 0 \\ G, E, S_0, \_ \vdash e_1 : v_1, S_1, \_ \\ \vdots \\ G, E, S_{n-1}, \_ \vdash e_n : v_n, S_n, \_ \\ l_{xi} = newloc(S_n), \text{ for } i = 1, \ldots, n \text{ and each } l_{xi} \text{ is distinct} \\ l_{yi} = newloc(S_n), \text{ for } i = 1, \ldots, k \text{ and each } l_{yi} \text{ is distinct} \\ E' = E_f[l_{x1}/x_1] \ldots [l_{xn}/x_n][l_{y1}/y_1] \ldots [l_{yk}/y_k] \\ G, E', S_n, \_ \vdash e'_1 : v'_1, S_n, \_ \\ \vdots \\ G, E', S_n, \_ \vdash e'_k : v'_k, S_n, \_ \\ S_{n+1} = S_n[v_1/l_{x1}] \ldots [v_n/l_{xn}][v'_1/l_{y1}] \ldots [v'_k/l_{yk}] \\ G, E', S_{n+1}, \_ \vdash b_{body} : \_, S_{n+2}, R \\ R' = \begin{cases} None, \text{ if } R \text{ is } \_ \\ R, \text{otherwise} \end{cases} \end{array}}{G, E, S_0, \_ \vdash f(e_1, \ldots, e_n) : R', S_{n+2}, \_} \quad [\text{INVOKE}]$$

The function invocation rule specifies the following operations. First, the function's value is fetched from the current store. Second, the arguments to the function call are evaluated in left-to-right order. Then, new locations are allocated for the function's formal parameters, local variables and nested functions. A new environment $E'$ is created for the function call, which maps the formal parameters, local variables, and the names of nested functions to their corresponding locations. The store $S_{n+1}$ maps these locations to their corresponding arguments, initial values, and function values respectively. Finally, the body of the function is evaluated with this new environment $E'$ and initial state $S_{n+1}$. The function invocation expression evaluates to the value returned by the function body, or the value `None` if the function body was completely evaluated without encountering a `return` statement.

A function value is created when evaluating a function definition. The rule is exactly the same for the definition of a globally defined function, a nested function, or a method defined in a class:

$$\frac{\begin{array}{l} g_1, \ldots, g_L \text{ are the variables explicitly declared as global in } f \\ y_1 = e_1, \ldots, y_k = e_k \text{ are the local variables and nested functions defined in } f \\ E_f = E[G(g_1)/g_1] \ldots [G(g_L)/g_L] \\ v = (x_1, \ldots, x_n, y_1 = e_1, \ldots, y_k = e_k, b_{body}, E_f) \end{array}}{G, E, S, \_ \vdash \texttt{def } f(x_1 : T_1, \ldots, x_n : T_n) \texttt{ -> } T_0 : b : v, S, \_} \quad [\text{FUNC-METHOD-DEF}]$$

The function value captures the environment $E$ in which it is defined. This allows a nested function to refer to local variables and functions defined in enclosing scopes. The captured environment $E$ is slightly modified as $E_f$, which overrides the mapping for variables explicitly declared as global within the function's body using the `global` declaration. This modification allows a nested function to reference a global variable even if there exists a local variable defined with the same name in an enclosing scope.

Next, we tackle dynamic dispatch of methods:

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e_0 : v_0, S_0, \_ \\ v_0 = X(a_1 = l_1, \ldots, f = l_f, \ldots, a_m = l_m) \\ S_0(l_f) = (x_0, x_1, \ldots, x_n, y_1 = e'_1, \ldots, y_k = e'_k, b_{body}, E_f) \\ n, k \geq 0 \\ G, E, S_0, \_ \vdash e_1 : v_1, S_1, \_ \\ \vdots \\ G, E, S_{n-1}, \_ \vdash e_n : v_n, S_n, \_ \\ l_{xi} = newloc(S_n), \text{ for } i = 0, \ldots, n \text{ and each } l_{xi} \text{ is distinct} \\ l_{yi} = newloc(S_n), \text{ for } i = 1, \ldots, k \text{ and each } l_{yi} \text{ is distinct} \\ E' = E_f[l_{x0}/x_0] \ldots [l_{xn}/x_n][l_{y1}/y_1] \ldots [l_{yk}/y_k] \\ G, E', S_n, \_ \vdash e'_1 : v'_1, S_n, \_ \\ \vdots \\ G, E', S_n, \_ \vdash e'_k : v'_k, S_n, \_ \\ S_{n+1} = S_n[v_0/l_{x0}] \ldots [v_n/l_{xn}][v'_1/l_{y1}] \ldots [v'_k/l_{yk}] \\ G, E', S_{n+1}, \_ \vdash b_{body} : \_, S_{n+2}, R \\ R' = \begin{cases} None, \text{ if } R \text{ is } \_ \\ R, \text{otherwise} \end{cases} \end{array}}{G, E, S, \_ \vdash e_0.f(e_1, \ldots, e_n) : R', S_{n+2}, \_} \quad [\textsc{dispatch}]$$

The rule for dynamic dispatch is similar to the rule for function invocation, with two main differences: (1) the target method is determined by first evaluating the object expression and then retrieving the function value that the method slot in the resulting object maps to; (2) the object itself is passed as the first argument to the method, before any of the arguments in the method-call expression.

The rules for accessing class attributes are relatively simpler:

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e : v_1, S_1, \_ \\ v_1 = X(a_1 = l_1, \ldots, id = l_{id}, \ldots, a_m = l_m) \\ v_2 = S_1(l_{id}) \end{array}}{G, E, S_0, \_ \vdash e.id : v_2, S_1, \_} \quad [\textsc{attr-read}]$$

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e_2 : v_r, S_1, \_ \\ G, E, S_1, \_ \vdash e_1 : v_l, S_2, \_ \\ v_l = X(a_1 = l_1, \ldots, id = l_{id}, \ldots, a_m = l_m) \\ S_3 = S_2[v_r/l_{id}] \end{array}}{G, E, S_0, \_ \vdash e_1.id = e_2 : v_r, S_3, \_} \quad [\textsc{attr-assign-expr}]$$

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e_2 : v_r, S_1, \_ \\ G, E, S_1, \_ \vdash e_1 : v_l, S_2, \_ \\ v_l = X(a_1 = l_1, \ldots, id = l_{id}, \ldots, a_m = l_m) \\ S_3 = S_2[v_r/l_{id}] \end{array}}{G, E, S_0, \_ \vdash e_1.id = e_2 : \_, S_3, \_} \quad [\textsc{attr-assign-stmt}]$$

Note that in the last two rules related to attribute assignment, the expression on the right-hand side is evaluated before the expression on the left-hand side. The only difference between an attribute assignment expression and an attribute assignment statement is that the latter does not evaluate to a value.

The last rule related to objects is that of object instantiation. This is the only rule that uses the syntax for class definitions described in Section 6.3:

$$\frac{\begin{array}{l} class(T) = (a_1 = e_1, \ldots, a_m = e_m) \qquad m \geq 1 \\ l_{ai} = newloc(S), \text{ for } i = 1, \ldots, m \text{ and each } l_{ai} \text{ is distinct} \\ v_0 = T(a_1 = l_{ai}, \ldots, a_m = l_{am}) \\ G, G, S, \_ \vdash e_1 : v_1, S, \_ \\ G, G, S, \_ \vdash e_2 : v_2, S, \_ \\ \vdots \\ G, G, S, \_ \vdash e_m : v_m, S, \_ \\ S_1 = S[v_1/l_{a1}] \ldots [v_m/l_{am}] \\ l_{init} = l_{ai} \text{ such that } a_i = \texttt{\_\_init\_\_} \\ S_1(l_{init}) = (x_0, y_1 = e'_1, \ldots, y_k = e'_k, b_{body}, E_f) \qquad k \geq 0 \\ l_{x0} = newloc(S_1) \\ l_{yi} = newloc(S_1), \text{ for } i = 1, \ldots, k \text{ and each } l_{yi} \text{ is distinct} \\ E' = E_f[l_{x0}/x_0][l_{y1}/y_1] \ldots [l_{yk}/y_k] \\ G, E, S_1, \_ \vdash e'_1 : v'_1, S_1, \_ \\ \vdots \\ G, E, S_1, \_ \vdash e'_k : v'_k, S_1, \_ \\ S_2 = S_1[v_0/l_{x0}][v'_1/l_{y1}] \ldots [v'_k/l_{yk}] \\ G, E', S_2, \_ \vdash b_{body} : \_, S_3, \_ \end{array}}{G, E, S, \_ \vdash T() : v_0, S_3, \_} \quad [\text{NEW}]$$

This rule performs the following operations. First, a new object $v_0$ with class $T$ is created by allocating locations for each attribute and method that is defined or inherited by class $T$. Second, the attribute initializers and method definitions are evaluated using the *global environment*; this distinction is important since method definitions do not capture the environment $E$ in which the object is being constructed. Third, a new store $S_1$ is created by modifying the current store $S$ with mappings for the newly allocated attributes and methods of $v_0$. Finally, the `__init__` method of the object $v_0$ is invoked via dynamic dispatch. The steps required to invoke this method are similar to that of general dynamic dispatch, with the following exceptions: (1) the `__init__` method does not have accept any arguments apart from the object on which it is invoked, (2) the `__init__` method does not contain any `return` statements and therefore does not return a value.

List objects are constructed in list-literal expressions where each sub-expression is evaluated in left-to-right order.

$$\frac{\begin{array}{l} n \geq 0 \\ G, E, S_0, \_ \vdash e_1 : v_1, S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : v_2, S_2, \_ \\ \vdots \\ G, E, S_{n-1}, \_ \vdash e_n : v_n, S_n, \_ \\ l_i = newloc(S_n), \text{ for } i = 1, \ldots, n \text{ and each } l_i \text{ is distinct} \\ v = [l_1, l_2, \ldots, l_n] \\ S_{n+1} = S_n[v_1/l_1][v_2/l_2] \ldots [v_n/l_n] \end{array}}{G, E, S_0, \_ \vdash \texttt{[}\ e_1, e_2, \ldots, e_n\ \texttt{]}\ : v, S_{n+1}, \_} \quad [\text{LIST-LITERAL}]$$

The rules for list selection, concatenation, and element update all use the locations corresponding to list elements and the values they map to in the store.

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e_1 : v_1, S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : int(i), S_2, \_ \\ v_1 = [l_1, l_2, \ldots, l_n] \\ 0 \leq i < n \\ v_2 = S_2(l_{i+1}) \end{array}}{G, E, S_0, \_ \vdash e_1[e_2] : v_2, S_2, \_} \quad [\text{LIST-SELECT}]$$

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e_1 : v_1, S_1, \_ \\ G, E, S_1, \_ \vdash e_2 : v_2, S_2, \_ \\ v_1 = [l_1, l_2, \ldots, l_n] \\ v_2 = [l'_1, l'_2, \ldots, l'_m] \\ n, m \geq 0 \\ l''_i = newloc(S_2), \text{ for } i = 1, \ldots, (m+n) \text{ and each } l''_i \text{ is distinct} \\ v_3 = [l''_1, l''_2, \ldots, l''_{n+m}] \\ S_3 = S_2[S_2(l_1)/l''_1] \ldots [S_2(l_n)/l''_n][S_2(l'_1)/l''_{n+1}] \ldots [S_2(l'_m)/l''_{n+m}] \end{array}}{G, E, S_0, \_ \vdash e_1 \texttt{ + } e_2 : v_3, S_3, \_} \quad [\text{LIST-CONCAT}]$$

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e_3 : v_r, S_1, \_ \\ G, E, S_1, \_ \vdash e_1 : v_l, S_2, \_ \\ G, E, S_2, \_ \vdash e_2 : int(i), S_3, \_ \\ v_l = [l_1, l_2, \ldots, l_n] \\ 0 \leq i < n \\ S_4 = S_3[v_r/l_{i+1}] \end{array}}{G, E, S_0, \_ \vdash e_1[e_2] = e_3 : v_r, S_4, \_} \quad [\text{LIST-ASSIGN-EXPR}]$$

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e_3 : v_r, S_1, \_ \\ G, E, S_1, \_ \vdash e_1 : v_l, S_2, \_ \\ G, E, S_2, \_ \vdash e_2 : int(i), S_3, \_ \\ v_l = [l_1, l_2, \ldots, l_n] \\ 0 \leq i < n \\ S_4 = S_3[v_r/l_{i+1}] \end{array}}{G, E, S_0, \_ \vdash e_1[e_2] = e_3 : \_, S_4, \_} \quad [\text{LIST-ASSIGN-STMT}]$$

Note that in the last two rules, the expression on the right-hand side of the assignment operator is evaluated before the expressions on the left-hand side.

The predefined functions `print` and `input` perform IO.

$$\frac{\begin{array}{l} G, E, S, \_ \vdash e : v, S_1, \_ \\ v = int(i) \text{ or } v = bool(b) \text{ or } v = str(n, s) \end{array}}{G, E, S, \_ \vdash print(e) : None, S_1, \_} \quad [\text{PRINT}]$$

$$\frac{\begin{array}{l} s \text{ is a user-provided input string of length } n \\ v = str(n, s) \end{array}}{G, E, S, \_ \vdash input() : v, S, \_} \quad [\text{INPUT}]$$

The predefined function `len` retrieves the length of a list or a string.

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e : v, S_1, \_ \\ v = [l_1, l_2, \ldots, l_n] \\ n \geq 0 \end{array}}{G, E, S_0, \_ \vdash \mathit{len}(e) : \mathit{int}(n), S_1, \_} \quad [\text{LEN-LIST}]$$

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e : v, S_1, \_ \\ v = \mathit{str}(n, s) \end{array}}{G, E, S_0, \_ \vdash \mathit{len}(e) : \mathit{int}(n), S_1, \_} \quad [\text{LEN-STR}]$$

The rules for `for` loops on lists and strings update the store after each iteration to map the loop variable's location to a value corresponding to a list element or substring respectively. In each case, we have a special version for early termination due to a return from the loop body.

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e : v, S'_0, \_ \\ v = [l_1, l_2, \ldots, l_n] \\ n \geq 0 \\ l_{id} = E(id) \\ S_1 = S'_0[S'_0(l_1)/l_{id}] \\ G, E, S_1, \_ \vdash b : \_, S'_1, \_ \\ S_2 = S'_1[S'_1(l_2)/l_{id}] \\ G, E, S_2, \_ \vdash b : \_, S'_2, \_ \\ \vdots \\ S_n = S'_{n-1}[S'_{n-1}(l_n)/l_{id}] \\ G, E, S_n, \_ \vdash b : \_, S'_n, \_ \end{array}}{G, E, S_0, \_ \vdash \texttt{for}\ id\ \texttt{in}\ e\ \texttt{:}\ b : \_, S'_n, \_} \quad [\text{FOR-LIST}]$$

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e : v_l, S'_0, \_ \\ v_l = [l_1, l_2, \ldots, l_n] \\ n \geq 0 \\ l_{id} = E(id) \\ S_1 = S'_0[S'_0(l_1)/l_{id}] \\ G, E, S_1, \_ \vdash b : \_, S'_1, \_ \\ S_2 = S'_1[S'_1(l_2)/l_{id}] \\ G, E, S_2, \_ \vdash b : \_, S'_2, \_ \\ \vdots \\ S_k = S'_{k-1}[S'_{k-1}(l_k)/l_{id}] \\ G, E, S_k, \_ \vdash b : \_, S'_k, R \\ 1 \leq k \leq n \qquad R \text{ is not } \_ \end{array}}{G, E, S_0, \_ \vdash \texttt{for}\ id\ \texttt{in}\ e\ \texttt{:}\ b : \_, S'_k, R} \quad [\text{FOR-LIST-RETURN}]$$

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e : v, S_0', \_ \\ v = str(n, c_1.c_2 \ldots c_n) \\ n \geq 0 \\ l_{id} = E(id) \\ S_1 = S_0'[str(1, c_1)/l_{id}] \\ G, E, S_1, \_ \vdash b : \_, S_1', \_ \\ S_2 = S_1'[str(1, c_2)/l_{id}] \\ G, E, S_2, \_ \vdash b : \_, S_2', \_ \\ \vdots \\ S_n = S_{n-1}'[str(1, c_n)/l_{id}] \\ G, E, S_n, \_ \vdash b : \_, S_n', \_ \end{array}}{G, E, S_0, \_ \vdash \texttt{for}\ id\ \texttt{in}\ e\ \texttt{:}\ b : \_, S_n', \_} \quad [\textsc{for-str}]$$

$$\frac{\begin{array}{l} G, E, S_0, \_ \vdash e : v_s, S_0', \_ \\ v_s = str(n, c_1.c_2 \ldots c_n) \\ n \geq 0 \\ l_{id} = E(id) \\ S_1 = S_0'[str(1, c_1)/l_{id}] \\ G, E, S_1, \_ \vdash b : \_, S_1', \_ \\ S_2 = S_1'[str(1, c_2)/l_{id}] \\ G, E, S_2, \_ \vdash b : \_, S_2', \_ \\ \vdots \\ S_k = S_{k-1}'[str(1, c_k)/l_{id}] \\ G, E, S_k, \_ \vdash b : \_, S_k', R \\ 1 \leq k \leq n \qquad R \text{ is not } \_ \end{array}}{G, E, S_0, \_ \vdash \texttt{for}\ id\ \texttt{in}\ e\ \texttt{:}\ b : \_, S_k', R} \quad [\textsc{for-str-return}]$$

Finally, the rule for evaluating a ChocoPy program involves first initializing a global environment and the initial store with globally defined variables and functions, and then evaluating the sequence of top-level statements. Let $\emptyset$ represent an empty mapping. Then, the top-level rule is:

$$\frac{\begin{array}{l} g_1 = e_1, \ldots, g_k = e_k \text{ are the global variable and function definitions in the program} \\ P \text{ is the sequence of statements in the program} \\ l_{gi} = newloc(\emptyset) \text{ for } i = 1, \ldots, k \text{ and each } l_{gi} \text{ is distinct} \\ G = \emptyset[l_{g1}/g_1] \ldots [l_{gk}/g_k] \\ G, G, \emptyset, \_ \vdash e_1 : v_1, \emptyset, \_ \\ \vdots \\ G, G, \emptyset, \_ \vdash e_k : v_k, \emptyset, \_ \\ S = \emptyset[v_k/l_{g1}] \ldots [v_k/l_{gk}] \\ G, G, S, \_ \vdash P : \_, S', \_ \end{array}}{\emptyset, \emptyset, \emptyset, \_ \vdash P : \_, S', \_} \quad [\textsc{program}]$$

When no valid rule can be applied to a given expression, the program aborts after printing an appropriate error message. Due to the myriad of semantic checks and typing rules enforced at compile-time, the set of errors that can occur at run-time is limited. The following is the standard set of run-time errors that can occur during the execution of a ChocoPy program:

1. Invalid argument (during invocation of `print` or `len`)

2. Division by zero

3. Index out of bounds (during string selection or list element access)
4. Operation on `None` (during method dispatch, attribute access, or list operations)
5. Out of memory (when allocating a new object)

The operational semantics do not specify what happens in the event of arithmetic integer overflow. This manual only specifies semantics for signed integer arithmetic that fits within 32 bits. Overflow is considered undefined behavior, and implementations may handle such situations in any manner of their choosing.

## 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. Countless typos were identified by the students taking CS164 at UC Berkeley in Fall 2018.

## A Known incompatibilities with Python

The following are the known cases where a valid ChocoPy program either does not correspond to a valid Python program, or has different semantics:

- The expression `True == not False` is valid syntax in ChocoPy, but not in Python. In Python, `not` cannot appear on the right side of a binary operator that has higher precedence than `not`, due to quirks in its grammar. In Python, the corresponding valid expression is `True == (not False)`.
- Python 3 does not allow forward references to classes that have not yet been defined. For example, the variable definition `x:A = None` is invalid in Python if this line appears before the class `A` has been completely defined. In ChocoPy, this is valid as long as the class `A` is defined anywhere in the program. A compromise is to use quoted type annotations: the syntax `x:"A" = None` is valid in both ChocoPy and Python, and has exactly the same meaning. In Python 3.7, forward references can be enabled by running `from __future__ import annotations`[2].
- ChocoPy has assignment expressions, while Python 3.7 does not[3]. Since ChocoPy uses the syntax of Python's chained assignment statements for assignment expressions, a statement of the form `a = b = c = d` has different evaluation order in ChocoPy and Python.
- Since integer overflow leads to undefined behavior in ChocoPy, there is no compatibility with Python when dealing with integer values less than $2^{31}$ or at least as large as $2^{31}$.

[2]PEP 563 mentions that forward references will be allowed by default in Python 4.0.
[3]PEP 572 proposes adding assignment expressions to Python 3.8.