CSPy Jabberwocky Documentation

Written by Lyndsay LaBarge ’17 and
Maya Montgomery ’18 in the summer of 2016,
under the supervision of Professor Alistair Campbell

A note on printing:
If you wish to print this documentation, you may want to consider printing only pages 1 – 27, as the remaining 33 pages contain the auto-generated CSPy grammar rules, descriptions of the built-in type system and functions, and descriptions of the type-checking functions. While hopefully helpful, this very detailed information is perhaps not crucial to have on hand. Please help save the trees! Print double-sided!

Contents:
This documentation describes the implementation of CSPy, a dialect of Python designed and written by Hamilton College students. Specifically, this refers to the “Jabberwocky” version of CSPy (i.e. the first functional version, with the ability to run CSPy programs), developed in the summer of 2016 by Lyndsay LaBarge ‘17 and Maya Montgomery ‘18 and based on a solid foundation written by Eric Collins ‘16 and Alex Dennis ‘18 in the summer of 2015. This documentation is intended to assist future programmers in understanding CSPy behind-the-scenes.

Language purpose:
CSPy is a statically typed programming language meant for computer science beginners. It aims to encourage conscientious coding by forcing the user to carefully specify their intentions. For example, CSPy mandates the specific declaration of variable types and disallows automatic type conversions. CSPy attempts to create an environment in which users fully understand their code.

Pages

CSPy Master Documentation (cspy_master.py)

Below is a description of the entire process by which CSPy is analyzed, type-checked, translated, and executed. The file `cspy_master.py` conducts this process. The current usage command: `python cspy_master.py filename.cspy`

1. The **master** sets up the lexer and parser with the rules defined in the **lexer** and **parser** files, then executes first the lexer, then the parser, on the given CSPy file. (This project uses PLY, an implementation of lex and yacc parsing tools for Python). This returns a parse tree (type: AST, defined in **data_struct**).
2. The **master** checks the parse tree for imported files. At this point, the **master** now begins a new compilation process (steps 1 - 5) for each CSPy file being imported, up until but not including runtime. Once processed, the **master** uses the returned parse tree of each imported file to add the appropriate methods and attributes to the parse tree of the parent file.
3. The **master** passes the parse tree to **genenv**, which adds environments in proper scope to appropriate nodes in the tree.
4. The **master** passes the parse tree to **type_checker**, which checks the contents of the tree for type errors. If an error is found, a personalized message is displayed. Otherwise, the **master** passes the parse tree to **translate**.
5. **Translate** translates the CSPy line by line into Python 2.7, writing the code to a new file. It also writes a file containing a dictionary that maps the Python file line numbers to the CSPy file line numbers, for error reporting at runtime.
6. Finally, **master** calls **runtime**, which executes the Python file. Any runtime errors are caught and displayed in a slightly simplified manner, also substituting line numbers from the original CSPy file. Once the file is done executing or an error report is finished, all Python executables and line map files are deleted, so the CSPy user is left with only the CSPy file.

Additional Notes

genenv, **type_checker**, and **translate** all import and make use of **builtins** and **data_struct** in order to create and edit the parse tree and to type-check built-in functions and types.

If an error is found in a .cspy file, any .py or .txt files created in the compilation process must be removed before the program terminates. At any location with a planned system exit, such as in `type_error` in the **type_checker**, the function `remove_files` is called. The master writes the names of any imported .cspy files as well as the main .cspy file to a .txt file in the current working directory, so `remove_files` may read in the names of the files to search for. Note that this .txt file must be closed after writing each name, because if the file object is open when an error is found, `remove_files` will not be able to access the names. More information in runtime documentation.

CSPy Lexer Documentation (cspy_lexer.py)

For documentation on PLY (Python Lex-Yacc), please visit http://www.dabeaz.com/ply/ply.html.

PLY LEX AND TOKENS

A lexer is used to tokenize an input string. It splits a string into individual tokens. Tokens are usually given a name to indicate what they are.

According to the PLY model, all token identifiers must be contained within a list assigned to the variable ‘tokens’. In this sense, ‘tokens’ is a reserved word and cannot be used in any other context. Each token identifier corresponds to a variable or function whose identifier is prefixed with ‘`t_`’, another naming system specific to PLY. Anything that follows ‘`t_`’ must be a token identifier and a member of the tokens list.

`cspy_lexer.py` contains a multiple of variables and functions following the naming system outlined in the PLY documentation to define the tokens in the CSPy language.

NOTE: Some unused tokens are currently commented out because they are Python reserved words that have not yet been configured in CSPy.

TOKEN IDENTIFIERS AND REGULAR EXPRESSIONS

Each variable or function prefixed with ‘`t_`’ and followed by a token identifier is assigned a regular expression which corresponds to the token name. For example, any string matching the regular expression '`[0-9]+`' will be classified as an ‘`INTLITERAL`’ token:

```
t_INTLITERAL = r'[0-9]+'
```

Token definitions using variables are added to the lexer in order of decreasing regular expression length. This means ‘==’ will be added to lexer before ‘=’, and any string containing two equals signs will match the first regular expression, not the second.

Token definitions using functions are added to the lexer *before* definitions using variables and in the order they are listed in the lexer file.

The first line in a token definition must always be a regular expression. Consider the token definition for an ‘`IDENTIFIER`’ token below

```
def t_IDENTIFIER(t):
   r'[_a-zA-Z][a-zA-Z0-9_]*'
   t.type = reserved.get(t.value, 'IDENTIFIER')
   if t.value == 'True' or t.value == 'False':
       t.type = 'BOOLLITERAL'
   return t
```

Note that the first line of the function is a regular expression corresponding to an identifier in CSPy, which can begin with an underscore, a lowercase or uppercase alphabetic character, and can be followed by any number of underscores or alphanumeric characters.

The parameter ‘`t`’ is a LexToken object. LexToken objects have a type attribute, a value attribute, and a lexer attribute. The type attribute is the token identifier, e.g. ‘`INTLITERAL’`, and the value attribute is the input string corresponding to the identifier, e.g. ‘7’. The lexer attribute is the lexer the token has been tokenized by.

All token definitions using functions must return ‘`t`’ or else the token object will disappear once the function finishes executing.

RESERVED WORDS

In addition to the tokens list, `cspy_lexer.py` also contains a dictionary of reserved words whose keys are reserved CSPy words and whose values are token identifiers corresponding to their keys, e.g. `‘if’ :’IF’ , ‘else’ : ‘ELSE’`.

Unlike the rest of the token identifiers, reserved words do not have to corresponding ‘`t_`’ variables or functions. Any reserved word will match the regular expression for an identifier. The ‘`t_INDENTIFIER`’ function defined above will assign the type attribute of the LexToken to a reserved words token identifier if the value of the token is in the dictionary of reserved words. If the identifier is not a reserved word, the LexToken type attribute will simply be ‘`IDENTIFIER`’ .

IGNORE

`t_ignore` is a special token definition. It is a regular expression that specifies which characters can be ignored by lexer (usually whitespace). The CSPy lexer ignores space and tab characters that *do not* relate to line indentation, e.g. spaces between letters, etc.

```
t_ignore_WS =  r'[ \t]'
```

LINE NUMBERS

By default, the lexer does not keep track of new lines. The `lineno` attribute of the lexer must be updated manually whenever the lexer encounters a newline token. The CSPy lexer keeps track of line numbers by updating the lexer’s `lineno` attribute in `t_CONTLINE` and `t_pass_start`.

INDENTATION

Additional attributes for a PLY lexer can be created after a lexer object has been created. In `cspy_master.py`, the CSPy lexer is assigned two additional attributes, `indentstack` and `indentedline`. `indentstack` is a stack containing the indentation levels of the program, where the indentation level on the top of the stack is the current indentation level in the lexing process.

Indentation is handled by `t_indent_INDENT`.

ILLEGAL CHARACTERS

Whenever the lexer encounters illegal characters (like ‘`$`’), `t_error` is invoked and a syntax error message is displayed containing the CSPy line and line number, along with ‘`^`’s pointing to the illegal character(s). After displaying the error message, the lexer skips over the illegal character(s) and continues tokenizing the input stream.

CSPy Parser Documentation (cspy_parser.py)

For documentation on PLY (Python Lex-Yacc), please visit http://www.dabeaz.com/ply/ply.html.

SET UP

`cspy_parser.py` contains a multitude of functions whose names are prefixed with 'p_', as per the PLY model. Each function takes a single variable, p, which is a LexToken created by the CSPy Lexer (see lexer documentation for more details). The very first line of each function is a docstring, which corresponds to a grammar rule and uses the following format, where 'b', 'c', and 'd' are nonterminals or terminals that reduce to nonterminal 'a':

```
a : b c d
```

Multiple rules for the same nonterminal can be written within the same docstring using the following syntax:

```
a : b c d
  | e f g
```

The start rule for the grammar is the nonterminal "`file`", as specified by the variable '`start`'.

ABSTRACT SYNTAX TREE

Parsed input is stored in an abstract syntax tree (defined in `cspy_data_structs.py`; see data struct documentation for more details). For each function, the variable 'p' is an iterable whose indices correspond to a nonterminal or terminal in the grammar rule. For example:

```
  a   :   b      c      d
p[0]    p[1]   p[2]   p[3]
```

All non-terminals on the right hand side of the grammar rule evaluate to abstract syntax trees representing the expansion of said terminal. For example, if 'b' was a non-terminal, the value of `p[1]` would be an abstract syntax tree corresponding to the grammar rule '`b : l m n`' where '`l`', '`m`', and 'n' are terminals or nonterminals.

The value assigned to `p[0]` is the value which gets 'returned' by a parsing rule function. The majority of parsing functions assign `p[0]` to an abstract syntax tree, e.g. `p[0] = ast(p, label, children*)` where 'p' is itself, '`label`' is a string which is the identifier of the abstract syntax tree node, and `children` are the indices of p that need to be stored in the abstract syntax tree.

From the above example, if you wanted to store the value of 'b' and 'd', but not 'c' in an abstract syntax tree, you would write the following line of code:

```
p[0] = ast(p, “A NODE”, 1, 3)
```

ERROR REPORTING

In addition to containing the grammar rules for the CSPy language, `cspy_parser.py` also contains additional grammar rules which contain the special ‘`error`’ token, which accounts for the possibility of syntax errors. Use of this token allows the parser to recover and resynchronize itself to continue parsing the remainder of a CSPy program after encountering a syntax error. This process is described in detail in the PLY documentation, under the section ‘Recovery and synchronization with error rules’. A simple example, taken from the CSPy grammar, is described below.

```
def p_declaration_error(p):
   '''declaration : IDENTIFIER COLON error EQUALS expression '''
   print("invalid type\n")
```

A variable can either be declared or declared and initialized simultaneously. The above rule corresponds to the latter. A variable declaration is defined to be an identifier followed by a colon and a type identifier. An equals sign followed by an expression signifies a variable initialization. ‘`x:int = 4`’ is an example of a valid declaration with an initialization step that contains no syntax errors.

‘`x:7 = 7`’ is clearly not a valid variable declaration, as both ‘7’s are classified as integer literals by the parser. What follows the colon must be a type identifier, such as ‘`int`’ or ‘`bool`’. In the case that what follows the colon is not a valid identifier or is a reserved word, in the above example, everything following the colon up to the equals sign (‘7’) will be matched to the special ‘`error`’ token and the following actions will be taken:

1. `p_error`, the parsing error message function, will be invoked with the ‘`error`’ token as its sole argument
2. `p_error` will display the CSPy line and line number the error occurred on along with ‘^’s pointing to the error and a message identifying it as a syntax error
3. The parser will exit from `p_error` and `IDENTIFIER COLON error EQUALS expression` will reduce to `declaration`, invoking `p_declaration_error`, which will display the message “invalid type” to elaborate on the nature of the syntax error
4. The error token will go away and the parser will attempt to continue parsing the CSPy program from the LexTokens which follow expression

Note that the ‘`error`’ token should never appear on the end of the right hand side of a grammar rule, as it will make resynchronization more difficult once the rule is reduced. For more information and examples, see the PLY documentation.

PRECEDENCE

`cspy_parser.py` contains a tuple named ‘`precedence`’ which lists the precedence of specific tokens. Tokens are listed in precedence order of lowest to highest. Each entry in the precedence list is also a tuple whose first element is a string corresponding to the associativity of the token(s). The remaining elements in the tuple are the names of the token(s). Consider the following two entries from the precedence list

```
('left', 'PLUS', 'MINUS'),
('left', 'TIMES', 'DIVIDE', 'MODULO', 'INTDIV')
```

Because they are listed below the ‘`PLUS`’ and ‘`MINUS`’ tokens, ‘`TIMES`’, ‘`DIVIDE`’, ‘`MODULO`’, and ‘`INTDIV`’ have higher precedence. All six of these tokens are left-associative.

OUTPUT

A CSPy program with no syntax errors will produce a single abstract syntax tree. The parser also produces the following files each time changes are made to the grammar, which are automatically generated:

- `parser.out`
    - Contains a written version of the grammar described in `cspy_parser.py` and the parsing table as well as any S/R or R/R conflicts if they exist. Text file for personal use, debugging, etc.

- `parsetab.py`
    - A python version of the PLY parsing table for use during the parsing process. <u>DO NOT</u> edit.

THE LANGUAGE

There are currently almost 300 CSPy grammar rules, automatically generated by the parser and stored in the file `parser.out`. These can be found in the grammar rules documentation.

CSPy Data Structures Documentation (cspy_data_struct.py)

`cspy_data_struct.py` contains class definitions for the following:

- AST (abstract syntax tree)
- DeclarationException
- NotYetDeclaredException
- SignatureException

As well as the following global variables:

- `binary_overload`: Dictionary which associates binary operators to the names of their corresponding binary overload functions
- `unary_overload`: Dictionary which associates unary operators to the names of their corresponding unary overload functions
- `holds_env`: List containing labels of all AST nodes that contain environments

class AST : an abstract syntax tree node class

Attributes

label:string

The name of the node. See parser defs for node names (e.g. '`INTLITERAL`').

type:type_obj

The type of the node. Defaults to `None`. The type of the node is altered by the function *det_type* (found in `cspy_type_checker.py`), which sets the type attributes for all of the nodes in the AST.

children:list of ast

A list of all the children of the current node. Children are usually abstract syntax trees but may occasionally be strings. Children can be accessed through the overloaded indexing operator (`n.children[0]` is equivalent to `n[0]`).

parent:ast

The parent of the current abstract syntax tree node (the node in the tree which contains the current node as a child). All nodes have a parent except for the root of the tree, whose parent is `None`.

env:dict of [string|type_obj]

A dictionary representing the environment contained by the current AST node. Only nodes whose labels are in "`holds_env`" will have an *env* attribute defined.

lineNum:int

The number in the CSPy source file indicating where the code this node holds resides.

endLineNum:int

The number in the CSPy source file indicating where the code this node holds ends.

position:int

The index of the first character of code from the CSPy source file the current node holds.

endPosition:int

The index of the last character of code from the CSPy source file the current ast node holds.

column:int

The index of the first character of CSPy code the current node holds with respect to the line number the code is one. The function "`set_column_node (sourceCode)`" must be called on the root of the tree in order to initialize this attribute.

endColumn:int

The index of the last character of CSPy code the current node holds with respect to the line number the code is one. The function "`set_column_node (sourceCode)`" must be called on the root of the tree in order to initialize this attribute.

line:int

The line of CSPy where the code contained within the current node is found.

Methods

*__init__(p:YaccProduction, label:string, *children:int)*

Constructor for an AST node. Receives a "YaccProduction" `p` which is the parsing symbol the AST represents, a string `label` which is the name and type of the node, and a tuple of integers `children` which are the indices of `p` that should be added to the current node's `children` attribute.

set_column_num(s:string)

Sets the values of the `column`, `endColumn`, and `line` attributes for the current node and for all of the children of the current node. Receives a string `s` which is the CSPy source code.

add_children(children:list of int, p:YaccProduction)

Given a list of integers `children` which are the indices of `p` to be added to the children of the current node.

lookup_var(var:string) -> type_obj or [type_obj]

Looks up `var`, the name of the variable being looked up, and returns the type object or a list of type objects (in the case of overloaded functions or procedures) if `var` has been declared in the node's current scope (or its parent scopes). If the variable does not exist, a `NotYetDeclaredException` is raised.

`initiate_var(var:string, typ:type_obj)`

Given a string `var`, the name of the variable being initialized, and `typ`, the type of `var`, adds `var` to the current node's environment. If the variable already exists, its value is not a function or procedure, or `typ` is not a function or procedure, a `Declaration Exception` is raised. If the variable already exists and its value is a function or procedure, if `typ` has the same signature as its values, a `SignatureException` is raised.

`flatten(label:string) -> list of ast`

Flattens the current tree and returns a list of tree nodes whose `label` attribute is `label`.

`__getitem__(index:int) -> ast`

Overloads the indexing operator for an AST. Returns the AST from the current AST's `children` attribute whose index is `index`.

`__setitem__(index:int, value:ast)`

Overloads the indexing assignment operator for an abstract syntax tree. Sets the value of current AST's `children` attribute at `index` to `value`.

`__repr__() -> string`

Returns a string representation of the current abstract syntax tree.

EXCEPTIONS

DeclarationException

Raised if a variable declaration fails.

NotYetDeclaredException

Raised if a variable has not been declared.

SignatureException

Raised if a function or procedure has already been declared with a given signature.

CSPy Generate Environments Documentation (cspy_genenv.py)

Description: `cspy_genenv.py` generates environments, assigning variables to their appropriate scopes, within an AST parse tree.

Detailed Process

Tree traversal:

The function `generate_environments` is called by the master program and is passed a parse tree. It calls `tree_pass`, which traverses the tree and delegates the environment building by calling functions based on the label of the current node; most of the functions in this file are named with the format g_`NODE`, where `NODE` is the label of a parse tree node. Only nodes pertaining to scope have functions in this file.

Node functions:

These functions take an AST node (n) as their argument. Each node function begins with a comment explaining the children of the received node:

```
def g_declaration(n):
   # 0: identifier; 1: type
```

→ note: `0` means `n[0]`; `1` means `n[1]`

Then each node function performs the appropriate tasks for its given node. The AST is edited to add objects to nodes that can hold environments. Some functions check for errors, usually when some object (a variable, a class, a function) has already been declared in the current scope and the user is attempting to declare it again.

Error reporting:

When an error is found, the imported function `type_error`, contained in `cspy_type_checker.py`, is called to display a formatted and educational error message. As the goal is to help beginning programmers learn, these messages are as descriptive yet simple as possible. `type_error` receives a message as a string, and at least one AST node. The node(s) passed to `type_error` holds the section of code that contains an error. Please see documentation on `cspy_type_checker.py` to read more about `type_error`.

CSPy Type Checker Documentation (cspy_type_checker.py)

Description: `cspy_type_checker.py` handles the semantic type-checking of a CSPy program via an abstract syntax tree whose environments have already been generated (see `cspy_genenv.py` documentation for more information).

Traversing a CSPy AST:

The main function, `det_type`, receives an abstract syntax tree. It traverses the tree, calling type checking functions based on the label of the current node. All of the type checking functions in this file are named with the format `s_NODE`, where `NODE` is the label of a parse tree node. This file contains additional helper functions as well, whose identifiers are not preceded by an 's_'.

Type Checking:

Each type checking function (prefixed with an 's_') receives an AST node (n) as its sole argument. The first line of every function is a comment with the indices and descriptions of the node's children (taken from `cspy_parser.py`):

```
def s_member(n):
   # 0: object; 1: attribute terminal
```

→ note: `0` means `n[0]`; `1` means `n[1]`

Each type checking function performs the tests appropriate for the given node. If there is a type error, the function calls `type_error`, an error reporting function which receives an error message (a string) and the tree node(s) where the error occurred.

For detailed information on the specific type requirements checked by each node function, see documentation on Type Checking Functions.

Error Reporting:

When a type error is found, `type_error` is called to display a detailed error message, containing the line and column number of the error, and a short description of what went wrong. These error messages are written for beginners and aim to use simple language to give the user helpful information about the error. The following occurs for every node passed to `type_error`:

1) The line and column number of the start of the code containing the error is displayed, along with the type of the node containing the error, if one exists.
2) The line of code from the source file containing error is output and "underlined" with the symbol "`^`", highlighting the portion of code within the line where the error occurred.

Finally, `type_error` displays the given error message.

For example, below is a CSPy program along with the error message for the type error it contains:

```
:: p : list of int = [1,2,3] ::

for item in p:
   print(item + "!")

--------------------------------------------------------

CSPy : Type Error
Line 4, Column 11: int
   print(item + "!")
         ^^^^
Line 4, Column 18: string
   print(item + "!")
                ^^^
The binary operator '+' is defined for the left-hand side (int),
but it does not have a signature matching the right-hand side
(string).
--------------------------------------------------------
```

CSPy Translator Documentation (cspy_translate.py)

Description: `cspy_translate.py` handles the translation of CSPy to Python 2.7 when given a type-checked parse tree.

Detailed Process

Set up:

The function `translate` is called by the master program and is passed a parse tree and the name of the CSPy file. Within the current working directory (ignoring any given path in the filename), it creates a file with the same filename but with the extension .py, then calls `toPython` on the parse tree to begin translation.

Tree traversal:

The function `toPython` traverses the tree and delegates translation by calling functions based on the label of the current node; most of the functions in this file are named with the format `c_NODE`, where `NODE` is the label of a parse tree node. This file contains additional helper functions as well, whose identifiers are not preceded by a ‘`c_`’.

Node functions:

These functions take three arguments: an AST node (`child`), a file object (`file`), and the current indentation level as measured by strings such as `"\t\t"` (`tabs` - set to a default of an empty string). Each node function begins with a comment explaining the children of the received node:

```
def c_MEMBER(child, file, tabs=""):
   # 0: identifier; 1: attribute name
```

→ note: `0` means `child[0]`; `1` means `child[1]`

Then each node function calls `toPython` on any appropriate children, and / or outputs Python code to the output file.

- e.g. when `toPython` sees a node labeled “`FILE`”, it will call `c_FILE`, which in turn calls `toPython` on all of its children (docstring, import block, declaration suite, and block) to be further broken down.
- e.g. when `toPython` sees a node labeled “`LITERAL_STRING`”, it simply writes the string to the output file because there is no more breaking down needed.

Line mapping:

At the end of every output with a new line (such as any single statement), the current line number in the output file is saved in a dictionary as the key to the current CSPy file line number. When translation is complete, a new file is created with the same filename plus “`_linemap.py`” (again, in the current working directory). The dictionary of the Python and CSPy line numbers is written to this file to be used for error reporting during runtime. See the documentation of [`cspy_runtime.py`] for more details.

Additional Notes

Irregular keywords:

This file includes a dictionary "`replace`" that holds a handful of specific keywords that need to be replaced when translating. For example, "&&" is a valid operator in CSPy, but must be replaced with "`and`" when translating to Python.

Global variables:

This file includes several global variables that are generally used in situations where a node function may need information that is not present in its received node. For example, `in_class` keeps track of whether or not the translator is currently writing a class definition; this variable is necessary in, for example, `c_DECLARATION_SUITE`, in order to decide between writing a normal series of variable declarations and writing an `__init__` method to declare class attributes. (More details on the global variable `last_var` in "**Class constructors**" below, and on `assign_me` in "**Overloaded functions**" below.)

Class constructors:

In CSPy, creating an instance of a user-defined class looks like:

```
myPet : Pet = Pet("Spot")
```

where the user-defined constructor is named after the class. In order to handle class attributes, this translator writes an `__init__` method consisting only of the class definition's main variable block (i.e. the class attributes).

So when a class instance is created - using the above example - the translator outputs `myPet = Pet()`, which will call the `__init__` to declare the class attributes, and then outputs `myPet.Pet("Spot")`, which will call the user-defined constructor as a method. (The line mapping is adjusted accordingly.)

However, the variable `myPet` is not passed to `c_CONSTRUCTOR_CALL`, and so a global variable, `last_var`, is used to access this identifier in order to output the call of the user-defined constructor.

Also note that a class may have multiple constructors defined (see "**Overloaded functions"** below).

Overloaded functions:

CSPy allows for overloaded function signatures, i.e. function definitions that share the same name but accept different parameters. (Note: though in CSPy terms a "function" returns a value and a "procedure" is void, in this case function is simply a general term; procedures may also be overloaded.) Of course, this means no two functions may share the same name *and* the same list of parameter types, as the functions are distinguished by their parameter type lists.

In translation, this overloading is handled by changing the names of the functions. When translating a function definition, if the value of the identifier in the node's parent environment is a list, then the identifier is associated with more than one function, and thus is overloaded. The name of each overloaded function is translated to the following format: `_funcname_params`. For example:

```
def myFunc (x:int)              →  _myFunc_int
def myFunc (x:string)           →  _myFunc_string
def myFunc (x:string, y:int)    →  _myFunc_string_int
```

When a function is called, the translator again checks if the function is overloaded. If it is, the translator uses the above established format to find the translated function name, but this time using the types of the given arguments instead of the defined parameter types. For example:

```
myFunc(6)          →   _myFunc_int(6)
myFunc("hi")       →   _myFunc_string("hi")
myFunc("hi", 3)    →   _myFunc_string_int("hi", 3)
```

In this way, all the overloaded functions are translated into their own separately named and callable functions in the Python file.

If a user is attempting to assign an overloaded function to a variable, the global variable `assign_me` comes into play. `assign_me` holds the name of the identifier to which a value is being assigned. In `c_VARIABLE`, the identifier's type is looked up (it matches that of an overloaded function) and its parameter type list is passed on to `overload_name` in order to assign the correct overloaded function to the variable.

Import readline:

`readline` is a module imported into each translated Python file. Adding this allows for the use of `input()` in .cspy files. Rather than determine whether or not a given file will require the module, the translator simply outputs this import statement to every Python executable. See `cspy_runtime.py` documentation on "**Running the file**" for a more detailed explanation.

CSPy Runtime Documentation (cspy_runtime.py)

Description: `cspy_runtime.py` runs the Python executable file as the final step in the compilation process, handles any runtime errors, and removes all the extraneous files which were created throughout the compilation process.

Detailed Process

Set up:

The function `run` is called by the master program and is passed the name of the CSPy file and a list of imported module names. It checks if the Python executable exists. If it doesn't, something unexpected has gone wrong somewhere in the compilation process, and `cspy_runtime.py` errors.

Running the file:

Many methods have been tested for this purpose, all with various pros and cons. Currently `run` is using `os.system` to execute the Python file. Though many sources say the `subprocess` module is a better choice, it does not appear to easily allow the function `input()` (more details in "**Other run methods**" below). `os.system` calls a bash command to run the Python executable and pipe any `stderr` (standard error) into a text file. Any intended output from the executable prints to the terminal, and any input during runtime is entered into the terminal.

Note: `os.system` also had some difficulty with `input()`, namely that it considered input prompts to be in the same category as `stderr` and therefore output these to the text file instead of the terminal. Research appears to show this is an unresolved bug. One forum coder's suggestion was to simply include `import readline` in the Python file. This miraculously works, allowing the use of `input()`, and so the translator currently imports `readline` into every Python executable. A messy fix, perhaps, and one that may have unforeseen consequences, but currently not a gift horse we're looking in the mouth.

Error reporting:

When an error is found, this file formats the error message to be more beginner-friendly. The error message is read in from the text file specified above.

In the traceback, every pair of lines consists of the file info and the appropriate line of code. (Note: All files present in the traceback should be Python files.) Below is an example Python error message straight from the terminal:

```
Traceback (most recent call last):
 File "ex.py", line 4, in <module>
   divide(6)
 File "ex.py", line 3, in divide
   y = x / 0
ZeroDivisionError: integer division or modulo by zero
```

A regular expression is used to extract the filename and the line number from the first line of each pair of lines in the traceback. Then, using the predetermined naming format `filename_linemap.py`, the Python-to-CSPy linemap dictionary created in `cspy_translate.py` is imported (more details in "**Line mapping**" below). This allows `run` to convert the extracted line number to the CSPy file's line number, to properly pinpoint the erroring code in the user's .cspy file.

Once all the lines have been processed, the given error message is printed, followed by the traceback. Below is the CSPy version of the above error message:

```
THERE IS AN ERROR IN FILE 'ex.cspy', LINE 4:
   y = x / 0

ZeroDivisionError: integer division or modulo by zero

TRACEBACK:
File 'ex.cspy', line 6:
   divide(6)
File 'ex.cspy', line 4:
   y = x / 0
```

Note: If there is an error but no traceback is found, something went wrong with the compilation process and not necessarily with the user's CSPy code.

Removing files:

Whether there's a runtime error or not, at the end this program removes the traces of compilation from the user's directory using the function `remove_files`. The master wrote a .txt file with the names of all .cspy files involved, which the function reads in. It looks for and removes the Python executable, the line map file, and any byte code files (.pyc ext) that remain. All files were output to the current working directory for ease of use, and so that is where this file looks when removing the extraneous files.

Additional Notes

Line mapping:

The current system for importing the line map dictionary is to simply write the dictionary to a file during translation, and then during runtime use `line_map = importlib.import_module(map_name)` to directly import the dictionary. However, this only works if the `_linemap.py` file is in the same directory as the running files, as Python's importing does not directly support paths. As the translator is currently outputting files to the current working directory, this seems to be working fine. But as it has caused issues in the past, the code with the messier but more regularly functional method remains in the file, commented out. Its explanation:

The current system for importing the line map dictionary is not ideal: the `_linemap.py` file, created during translation, consists of the dictionary and code that prints out each key and value of the dictionary. Then runtime creates a subprocess in which it runs the `_linemap.py` file, reads its output (the dictionary), and creates a new, local dictionary using the given keys and values, converting the strings to ints. This makes sure the `_linemap.py` file can be used no matter what directory it resides in (assuming that its path is known).

Other run methods:

Ignoring the aforementioned issue of `input()`, the best method found so far was to use the module `subprocess.Popen` to attempt to run the Python executable - as it sounds, this module creates a subprocess in which to execute its given command. If there was a runtime error, the error message was retrieved from the process - using the `Popen` method `communicate()` - and saved to a variable:

```
new_process = subprocess.Popen([‘python’, filename],
                               stderr = subprocess.PIPE)
error = new_process.communicate()[1]
```

The issue with this is that the function `input()`, which introductory students will likely use often to interact with their programs, will not work unless you explicitly use `communicate()` each time input is needed. It is not efficiently possible to plan for these `input()` calls. Other methods of `subprocess` besides `Popen` (such as `call` or `check_output`) may or may not be able to handle `input()`, but regardless do not allow piping the standard error and so do not allow the formatting and line number swapping which we require. Therefore, we chose to use `os.system` and a less sophisticated method of standard error piping.

CSPy Syntax Guide

VARIABLE DECLARATIONS

All variable declarations must occur inside of a variable block. Variable blocks can only occur at the top of a code block. Below is a variable block containing a declaration for the variable `x`, which is an integer:

```
:: x:int ::
```

All global variables must be declared in a variable block at the top of the file. All variable blocks are optional; writing an empty variable block (`:: ::`) will cause a syntax error. Variables blocks are allowed to span multiple lines provided there is a comma before the new line and no declarations are split up between lines (i.e. you must finish a declaration before adding a comma and a newline).

`x:int` → Variable declaration for an integer.

`x:float` → Variable declaration for a float.

`x:bool` → Variable declaration for a boolean value.

`x:string` → Variable declaration for a string.

`x:list of ?` → Variable declaration for a list, whose elements have type '?'. Lists are homogeneous.

`x:[?]` → Alternative syntax for a list declaration for convenience (easier to declare nested lists, e.g. `x:[[int]]` vs `x:list of list of int`).

`x:tuple of (t1 * t2 * …)` → Variable declaration for a tuple, whose first element has type `t1`, second element has type `t2`, etc. Tuples can be heterogeneous.

`x:(t1 * t2)` → Alternative syntax for a tuple declaration (easier to declare nested tuples or nested lists, e.g. `x:[(int * string)]` where `x` is a list of tuples).

`x:dict of [k|v]` → Variable declaration for a dictionary whose keys have type `k` and whose values have type `v`. A dictionary's keys are homogenous, as well as its values. The key type of a dictionary does NOT have to be the same as its value type.

`x:set of ?` → Variable declaration for a set whose members have type '?'. Sets are homogeneous.

`x:frozenset of ?` → Variable declaration for a frozenset whose members have type '?'. Frozensets are homogeneous.

`x:file` → Variable declaration of a file (type returned by the built-in function `open`).

`x:fn (p1, p2, …) -> r` → Variable declaration of a function whose first parameter has type `p1`, second has type `p2`, etc. and whose return type is `r`.

`x:proc (p1, p2, …)` → Variable declaration of a procedure whose first parameter has type `p1`, second has type `p2`, etc.

`x:Exception` → Variable declaration of an exception type.

`x:id` → Variable declaration for an instance of a user defined class `id`.

TYPE LITERALS

NOTE: There is no ‘`None`’ type in CSPy.

`int` - any integer eg. `7`

`float` - a decimal representation of a number eg. `5.2`

`bool` - `True` or `False`

`string` - `“Hello World”`, `‘Hello World’` CANNOT include new line(s)
`"""Hello World"""`, `'''Hello World'''` CAN include new line(s)

`list` - `[1, 2, 3, 4]`

`tuple` - `(“A”, 1)`

`dict` - `{ 1 : “a”, 2 : “b” }`

`set` - `{5, 6, 7, 8}`
NOTE: `{}` is an empty dictionary. To make an empty set, use `makeset()`.

`function` - `lambda (x:int, y:int) -> int : (x + y)`

There do not exist frozenset, procedure, or file literals in CSPy.

CLASS DEFINITIONS

```
class Circle:
     :: center:tuple of (int * int), radius:int ::

     def Circle(x:int, y:int, r:int):
        center = (x, y)
```

```
        radius = r

    def distance(c:Circle) -> int:
      ...........
```

All class attributes must be declared in a variable block above all of class methods, immediately underneath the line of code containing the `class` keyword. In the example above, `center` and `radius` are `Circle` attributes.

Instead of naming the class constructor `__init__`, all constructors must be procedures whose identifier corresponds to the class name. Constructors do not return anything. While they are classified by the parser as a procedure, constructors can only be called once per instance. In the example above, `Circle(int,int,int)` is a constructor.

The method `distance` takes a `Circle` object as a parameter. Classes are allowed to have local variables in their methods whose type corresponds to the class being defined, however they are not allowed to have them as attributes. For example, `Circle` cannot have an attribute `innerCircle:Circle` (see **Recommendations** for more details).

Class methods should not allow local variables or parameters with the same names as attributes, to avoid confusion with the user. Neither of these are currently caught by the type-checker, but should be. The parameters will be translated incorrectly.

Classes can have super classes. Below, `Shape` is a super class of `Circle`:

```
class Circle extends Shape:
```

You can access the methods and attributes of a class instance object the same way you would in Python (e.g. `myCircle.center`).

FUNCTION DEFINITIONS

```
def add(x:int, y:int) -> int:
     return x + y

def add(x:float, y:float) -> float:
     return x + y
```

The types of a function's arguments must be specified in the parameter list. The return type of a function is indicated by the type following a `->`. All of the return statements in a function must return a value whose type matches the function’s declared return type.

There can be multiple definitions for a single function, provided that their type signatures are all different. The return type is not considered part of the function signature, therefore the following additional definition of `add` would not be allowed:

```
def add(x:int, y:int) -> float:
     return tofloat(x + y)
```

PROCEDURE DEFINITIONS

```
def output(x:int):
     print(x)

def output(x:float):
     print(x)
```

The types of a parameter's arguments must be specified in the parameter list. Procedures do not return a value, and their return type does not need to be specified (i.e. there is no ‘`void`’ like there is in C++). If a procedure contains a non-empty return statement, an error will occur.

Like with functions, there can be multiple definitions for a single procedure, provided that their type signatures are all different.

FOR LOOP

For loops use the same syntax as Python. The only notable difference is you cannot iterate over a tuple because they are heterogeneous and can contain more than a single element type. You can have a variable block within a for loop. For example:

```
for i in [1,2,3,4]:
   :: x :int = 0 ::
   ……
```

CSPy also introduces a special for loop syntax, intended to replace Python’s `range()` method (though `range` is still functional in CSPy):

```
for i in 1..4:
```

The above is equivalent to the previous for loop or to `for i in range(1,5)`. This type of `range` is inclusive with its ending value, not exclusive like the `range()` method.

WHILE LOOP

While loops use the same syntax as Python. CSPy does not have the same truth testing system as Python, where any object can be tested for truth value. Currently, only boolean values, or operators and functions which return a boolean value, are allowed to be the condition of a while loop. This is the same for conditionals (`if`, `elif`, and the ternary operator). A while loop can have a variable block at the top of its code block.

CONDITIONALS

Conditionals use the same syntax as Python (`if-elif-else`). Conditionals blocks can have a variable block at the top of the code block.

STATEMENTS

A statement is any of the following. More than one simple statement can occur on the same line, provided they are separated from each other with a ‘`;`’ (semicolon).

Assignment

Note: CSPy does not currently support chained operations, e.g. `x = y = 5`

Variable assignment: `x = 4` (where `x` has already been declared as an int)

Indexing assignment: `myList[0] = 7` (where `myList` has already been declared as a `list of int`)

Slicing assignment: `myList[1:] = [6,7,8]` (where `myList` has already been declared as a `list of int`)

NOTE: Tuples can only be indexed or sliced using integer literals as indices, since it is impossible to tell what type an indexing or slicing operator would return from a heterogeneous tuple at compile time if any of the indices are variables.

```
x = myTuple[0]       ACCEPTABLE
x = myTuple[y]       UNACCEPTABLE
```

Augmented Assignment: `x+= 10` (where `x` has already been declared as a int)

Procedure Call

```
print(“Hello World”)
```

Return Statement

`return (x == y)` or `return`

Raise Exception

`raise` *`identifier`*

Identifier must be an exception type.

Delete Statement

`del` *`expression`*

Break, Pass, and Continue

`break` or `pass` or `continue`

TRY EXCEPT

Try-except clauses use the same syntax as Python. CSPy supports `try-except`, `try-except-else`, and `try-except-finally`. Each block in a try-except clause can contain its own variable declaration block. For except statements, if except is followed by an identifier, it must be a declared exception type.

EXPRESSIONS

Calculation: `6 + 7`

For a list of all of the binary and unary operators and the types that support them, please see the documentation for the built-in library.

Ternary Operator: `(x == 1) ? 1 : 0` Python's if-else ternary operator is not supported. CSPy has a C++ style ternary operator. The conditional of the ternary operator must a boolean value, a boolean expression, or a function which returns a boolean value. The type of the then clause must match the type of the else clause.

Function Call: `add(1,3)`

Grouping: `(x + 4) * 10`

Indexing: `x = myList[4]` (where `x` has already been declared as an int, and `myList` has already been declared as a list of int)

Slicing: `s = myString[1:]` (where `s` and `myString` have already been declared as type string)

Membership Testing: `3 in [1,2,3]` `3 not in [1,2,3]`

An error will occur if you try to test for membership in a container whose element type does not correspond to the sequence's type (e.g. `4.0 in [1,2,3]`) since this will always be false.

Identity Testing:

```
x is y
x is not y
```

An error will occur if you try to test for identity with two objects that have different types since this will always be false.

CSPy Future Development Recommendations

Importing Python modules: As students begin to master the basics of programming, they may require additional support through Python modules such as `random` or `sys`. It would be simple to add a syntax to specify if a module is from Python (as it should not be processed as a .cspy file), but the difficult part is adding the module's methods and attributes to the .cspy file's parse tree for generating environments and type-checking.

Developing graphical interface: A graphical interface can help ease students into the use of UNIX shells. A prototype was developed in the summer of 2014 but is incomplete. An Emacs syntax highlighter also exists, but is also incomplete and no longer matches the current version / expansiveness of CSPy anyway. Potential features of the GUI:

- file exploration (for saving, opening)
- text editor
    - CSPy syntax highlighting
    - keyboard shortcuts (same as Emacs) including cheat sheet
- embedded "terminal" in which to view program results, and a "run" button
- options to check / uncheck:
    - whether or not to automatically run standalone code in main / imports
    - display 1 error message at a time or all at once. Currently, for both parsing and type checking, only the first error is guaranteed to be accurate. Any following errors can be side effects of the previous. To display > 1 error message, delete exit(1) in `type_error` and `parse_error` functions

Non-Exhaustive Return Statements: It would be nice to warn beginners if they have written a function that contains non-exhaustive return statements. For example:

```
def foo(x:int) -> bool
   if x == 1:
      return True
   else:
      print( “X is not 1”)
```

Class Attributes: Currently, class definitions don't allow attributes whose type is the class being defined, e.g. if we are in the class definition of a `Pet`, and class `Pet` has an attribute named `friend` whose type is `Pet` (i.e. `friend:Pet`), the type-checker will get confused and error if you try to use any of `friend`'s methods or attributes in a method in the class definition (e.g. `friend.name`). This is because at the current point in the type-checking process, none of `Class Pet`'s methods have been defined. We haven't yet thought of a nice way to solve this problem.

You CAN however use the class as the type of parameters or local variables in a class method, so classes are still able to interact with other objects of their class. For example, `Pet` could have a method called `play`, which took a `Pet` as a parameter, and this would be fine because `Pet` has already been defined as a type and is stored in the global environment with its methods.

CSPy Parser Grammar Rules

Automatically generated by PLY into `parser.out`. See parser documentation for more details on parser grammar.

Last updated: 7.8.2016

```
Rule 0     S' -> file
Rule 1     file -> optdoc importblock declaration_suite nonempty_block
Rule 2     file -> optdoc importblock declaration_suite empty
Rule 3     empty -> <empty>
Rule 4     optdoc -> DOCSTRING NL
Rule 5     optdoc -> empty
Rule 6     importblock -> nonempty_importblock
Rule 7     importblock -> empty
Rule 8     nonempty_importblock -> singleimport
Rule 9     singleimport -> import_statement
Rule 10    nonempty_importblock -> nonempty_importblock singleimport
Rule 11    import_statement -> IMPORT IDENTIFIER NL
Rule 12    import_statement -> IMPORT IDENTIFIER AS IDENTIFIER NL
Rule 13    import_statement -> FROM IDENTIFIER IMPORT TIMES NL
Rule 14    import_statement -> FROM IDENTIFIER IMPORT importlist NL
Rule 15    importlist -> IDENTIFIER
Rule 16    importlist -> IDENTIFIER AS IDENTIFIER
Rule 17    importlist -> importlist COMMA importlist
Rule 18    declaration_suite -> variableblock classblock methodblock
Rule 19    variableblock -> COLONCOLON nonempty_variableblock
           COLONCOLON NL
Rule 20    variableblock -> empty empty
Rule 21    nonempty_variableblock -> declaration
Rule 22    nonempty_variableblock -> nonempty_variableblock COMMA
           nonempty_variableblock
Rule 23    declaration -> IDENTIFIER COLON type
Rule 24    declaration -> IDENTIFIER COLON type EQUALS expression
Rule 25    classblock -> class_definition classblock
Rule 26    classblock -> empty
Rule 27    class_definition -> CLASS IDENTIFIER opt_generic
           opt_extends COLON NL INDENT class_suite DEDENT
Rule 28    class_suite -> optdoc declaration_suite
Rule 29    opt_extends -> EXTENDS type
Rule 30    opt_extends -> empty empty
Rule 31    opt_generic -> LT genericlist GT
Rule 32    opt_generic -> empty empty
Rule 33    genericlist -> IDENTIFIER EXTENDS type
Rule 34    genericlist -> genericlist COMMA genericlist
Rule 35    methodblock -> subroutine_definition methodblock
Rule 36    methodblock -> empty
Rule 37    subroutine_definition -> function_definition
Rule 38    subroutine_definition -> procedure_definition
```

```
Rule 39    function_definition -> DEF IDENTIFIER LPAREN argumentlist
           RPAREN ARROW type COLON suite
Rule 40    procedure_definition -> DEF IDENTIFIER LPAREN argumentlist
           RPAREN COLON suite
Rule 41    argumentlist -> nonempty_argumentlist COMMA
           nonempty_defaultlist
Rule 42    argumentlist -> nonempty_argumentlist empty empty
Rule 43    argumentlist -> nonempty_defaultlist empty empty
Rule 44    argumentlist -> empty empty empty
Rule 45    nonempty_argumentlist -> IDENTIFIER COLON type
Rule 46    nonempty_argumentlist -> nonempty_argumentlist COMMA
           nonempty_argumentlist
Rule 47    nonempty_defaultlist -> nonempty_defaultlist COMMA
           nonempty_defaultlist
Rule 48    nonempty_defaultlist -> IDENTIFIER COLON type EQUALS
           expression
Rule 49    suite -> NL INDENT optdoc block DEDENT
Rule 50    suite -> statement_simple NL
Rule 51    block -> variableblock nonempty_block
Rule 52    nonempty_block -> statement_complex empty
Rule 53    nonempty_block -> statement_complex nonempty_block
Rule 54    statement_complex -> loop
Rule 55    statement_complex -> conditional
Rule 56    statement_complex -> try_except
Rule 57    statement_complex -> statement_multi NL
Rule 58    statement_complex -> statement_multi SEMICOLON NL
Rule 59    statement_multi -> statement_multi SEMICOLON
           statement_simple
Rule 60    statement_multi -> statement_simple
Rule 61    statement_simple -> assignment
Rule 62    statement_simple -> procedure_call
Rule 63    statement_simple -> return
Rule 64    statement_simple -> assert
Rule 65    statement_simple -> CONTINUE
Rule 66    statement_simple -> BREAK
Rule 67    statement_simple -> PASS
Rule 68    statement_simple -> raise
Rule 69    statement_simple -> delete
Rule 70    raise -> RAISE IDENTIFIER
Rule 71    delete -> DEL expression
Rule 72    loop -> while_loop
Rule 73    loop -> for_loop
Rule 74    while_loop -> WHILE expression COLON suite
Rule 75    for_loop -> FOR IDENTIFIER IN expression COLON suite
Rule 76    for_loop -> FOR IDENTIFIER IN expression DOTDOT expression
           COLON suite
Rule 77    conditional -> IF expression COLON suite
           conditional_extension
Rule 78    conditional_extension -> empty
Rule 79    conditional_extension -> ELIF expression COLON suite
           conditional_extension
Rule 80    conditional_extension -> ELSE COLON suite
```

```
Rule 81    try_except -> TRY COLON suite exceptlist_nonempty empty
           empty
Rule 82    try_except -> TRY COLON suite exceptlist_nonempty
           except_else empty
Rule 83    try_except -> TRY COLON suite exceptlist_nonempty empty
           except_finally
Rule 84    try_except -> TRY COLON suite exceptlist_nonempty
           except_else except_finally
Rule 85    try_except -> TRY COLON suite empty empty except_finally
Rule 86    except_simple -> EXCEPT COLON suite
Rule 87    except_alias -> EXCEPT IDENTIFIER AS IDENTIFIER COLON suite
           exceptlist
Rule 88    except_specific -> EXCEPT IDENTIFIER COLON suite exceptlist
Rule 89    except_else -> ELSE COLON suite
Rule 90    except_finally -> FINALLY COLON suite
Rule 91    exceptlist_nonempty -> except_simple
Rule 92    exceptlist_nonempty -> except_alias
Rule 93    exceptlist_nonempty -> except_specific
Rule 94    exceptlist -> except_simple
Rule 95    exceptlist -> except_alias
Rule 96    exceptlist -> except_specific
Rule 97    exceptlist -> empty
Rule 98    assignment -> indexing assignment_operator expression
Rule 99    assignment -> slicing assignment_operator expression
Rule 100   assignment -> variable assignment_operator expression
Rule 101   assignment -> member assignment_operator expression
Rule 102   assignment_operator -> EQUALS
Rule 103   assignment_operator -> PLUSEQU
Rule 104   assignment_operator -> MINUSEQU
Rule 105   assignment_operator -> TIMESEQU
Rule 106   assignment_operator -> DIVEQU
Rule 107   assignment_operator -> MODEQU
Rule 108   assignment_operator -> BITANDEQU
Rule 109   assignment_operator -> BITOREQU
Rule 110   assignment_operator -> BITXOREQU
Rule 111   assignment_operator -> LSHIFTEQU
Rule 112   assignment_operator -> RSHIFTEQU
Rule 113   assignment_operator -> POWEQU
Rule 114   assignment_operator -> INTDIVEQU
Rule 115   indexing -> expression LBRACKET expression RBRACKET
Rule 116   slicing -> expression LBRACKET expression COLON expression
           optslice RBRACKET
Rule 117   slicing -> expression LBRACKET empty COLON expression
           optslice RBRACKET
Rule 118   slicing -> expression LBRACKET expression COLON empty
           optslice RBRACKET
Rule 119   slicing -> expression LBRACKET empty COLON empty optslice
           RBRACKET
Rule 120   optslice -> empty empty
Rule 121   optslice -> COLON empty
Rule 122   optslice -> COLON expression
Rule 123   procedure_call -> expression LPAREN expressionlist RPAREN
```

```
Rule 124   return -> RETURN empty
Rule 125   return -> RETURN expression
Rule 126   assert -> assertnomessage
Rule 127   assert -> assertmessage
Rule 128   assertnomessage -> ASSERT expression
Rule 129   assertmessage -> ASSERT expression COMMA literal
Rule 130   type -> function_type
Rule 131   type -> procedure_type
Rule 132   type -> tuple_type
Rule 133   type -> list_type
Rule 134   type -> dictionary_type
Rule 135   type -> set_type
Rule 136   type -> frozenset_type
Rule 137   type -> generic_type
Rule 138   type -> IDENTIFIER
Rule 139   function_type -> FN LPAREN typelist RPAREN ARROW type
Rule 140   procedure_type -> PROC LPAREN typelist RPAREN
Rule 141   generic_type -> IDENTIFIER LT nonempty_typelist GT
Rule 142   typelist -> nonempty_typelist COMMA
           nonempty_default_typelist
Rule 143   typelist -> nonempty_typelist empty empty
Rule 144   typelist -> empty empty nonempty_default_typelist
Rule 145   typelist -> empty empty empty
Rule 146   nonempty_typelist -> type
Rule 147   nonempty_typelist -> nonempty_typelist COMMA
           nonempty_typelist
Rule 148   nonempty_default_typelist -> QMARK type
Rule 149   nonempty_default_typelist -> nonempty_default_typelist
           COMMA nonempty_default_typelist
Rule 150   tuple_type -> tupleof
Rule 151   tuple_type -> tupleparens
Rule 152   tupleof -> TUPLE OF LPAREN tuple_typelist RPAREN
Rule 153   tupleparens -> LPAREN tuple_typelist RPAREN
Rule 154   tuple_typelist -> nonempty_tuple_typelist
Rule 155   tuple_typelist -> empty
Rule 156   nonempty_tuple_typelist -> type
Rule 157   nonempty_tuple_typelist -> nonempty_tuple_typelist TIMES
           nonempty_tuple_typelist
Rule 158   list_type -> listof
Rule 159   list_type -> listbracket
Rule 160   listof -> LIST OF type
Rule 161   listbracket -> LBRACKET type RBRACKET
Rule 162   set_type -> SET OF type
Rule 163   frozenset_type -> FROZENSET OF type
Rule 164   dictionary_type -> DICT OF LBRACKET type BITOR type
           RBRACKET
Rule 165   expression -> calculation
Rule 166   expression -> function_call
Rule 167   expression -> grouping
Rule 168   expression -> literal
Rule 169   expression -> indexing
Rule 170   expression -> slicing
```

```
Rule 171   expression -> ternary
Rule 172   expression -> member
Rule 173   expression -> identity
Rule 174   expression -> membership
Rule 175   expression -> variable
Rule 176   calculation -> expression PLUS expression
Rule 177   calculation -> expression MINUS expression
Rule 178   calculation -> expression TIMES expression
Rule 179   calculation -> expression DIVIDE expression
Rule 180   calculation -> expression PERCENT expression
Rule 181   calculation -> expression INTDIV expression
Rule 182   calculation -> expression POW expression
Rule 183   calculation -> expression BITOR expression
Rule 184   calculation -> expression BITAND expression
Rule 185   calculation -> expression LSHIFT expression
Rule 186   calculation -> expression RSHIFT expression
Rule 187   calculation -> expression EQUALTO expression
Rule 188   calculation -> expression NEQUALTO expression
Rule 189   calculation -> expression LT expression
Rule 190   calculation -> expression LE expression
Rule 191   calculation -> expression GT expression
Rule 192   calculation -> expression GE expression
Rule 193   calculation -> expression REQUALTO expression
Rule 194   calculation -> expression BOOLOR expression
Rule 195   calculation -> expression BOOLAND expression
Rule 196   calculation -> expression OR expression
Rule 197   calculation -> expression AND expression
Rule 198   calculation -> expression CARET expression
Rule 199   expression -> MINUS expression
Rule 200   expression -> PLUS expression
Rule 201   expression -> TILDE expression
Rule 202   expression -> EXMARK expression
Rule 203   expression -> NOT expression
Rule 204   function_call -> expression LPAREN expressionlist RPAREN
Rule 205   expressionlist -> nonempty_expressionlist
Rule 206   expressionlist -> empty
Rule 207   nonempty_expressionlist -> expression
Rule 208   nonempty_expressionlist -> nonempty_expressionlist COMMA
           nonempty_expressionlist
Rule 209   grouping -> LPAREN expression RPAREN
Rule 210   literal -> INTLITERAL
Rule 211   literal -> FLOATLITERAL
Rule 212   literal -> BOOLLITERAL
Rule 213   literal -> STRINGLITERAL
Rule 214   literal -> DOCSTRING
Rule 215   literal -> function_literal
Rule 216   literal -> procedure_literal
Rule 217   literal -> tuple_literal
Rule 218   literal -> list_literal
Rule 219   literal -> dictionary_literal
Rule 220   literal -> set_literal
```

```
Rule 221   function_literal -> LAMBDA LPAREN argumentlist RPAREN ARROW
           type COLON LPAREN expression RPAREN
Rule 222   tuple_literal -> LPAREN tuplelist RPAREN
Rule 223   tuplelist -> nonempty_tuple
Rule 224   tuplelist -> empty
Rule 225   nonempty_tuple -> singletontuple
Rule 226   nonempty_tuple -> crosstuple
Rule 227   singletontuple -> nonempty_expressionlist COMMA
Rule 228   crosstuple -> nonempty_expressionlist
Rule 229   list_literal -> LBRACKET expressionlist RBRACKET
Rule 230   dictionary_literal -> LCURLY dictionarylist RCURLY
Rule 231   dictionarylist -> nonempty_dictionarylist
Rule 232   dictionarylist -> empty
Rule 233   nonempty_dictionarylist -> expression COLON expression
Rule 234   nonempty_dictionarylist -> nonempty_dictionarylist COMMA
           nonempty_dictionarylist
Rule 235   set_literal -> LCURLY nonempty_expressionlist RCURLY
Rule 236   variable -> IDENTIFIER
Rule 237   ternary -> expression QMARK expression COLON expression
Rule 238   member -> expression DOT IDENTIFIER
Rule 239   identity -> expression IS expression
Rule 240   identity -> expression ISNOT expression
Rule 241   membership -> expression IN expression
Rule 242   membership -> expression NOTIN expression
Rule 243   variableblock -> COLONCOLON error COLONCOLON NL
Rule 244   declaration -> IDENTIFIER COLON error EQUALS expression
Rule 245   class_definition -> CLASS IDENTIFIER opt_generic
           opt_extends error NL INDENT class_suite DEDENT
Rule 246   class_definition -> CLASS IDENTIFIER opt_generic
           opt_extends COLON NL INDENT error DEDENT
Rule 247   opt_generic -> LT error GT
Rule 248   function_definition -> DEF error LPAREN argumentlist RPAREN
           ARROW type COLON suite
Rule 249   function_definition -> DEF IDENTIFIER LPAREN argumentlist
           error ARROW type COLON suite
Rule 250   function_definition -> DEF IDENTIFIER LPAREN argumentlist
           RPAREN ARROW error COLON suite
Rule 251   procedure_definition -> DEF error LPAREN argumentlist
           RPAREN COLON suite
Rule 252   procedure_definition -> DEF IDENTIFIER LPAREN argumentlist
           error COLON suite
Rule 253   nonempty_argumentlist -> error COLON type
Rule 254   nonempty_defaultlist -> IDENTIFIER COLON error EQUALS
           expression
Rule 255   while_loop -> WHILE error COLON suite
Rule 256   for_loop -> FOR error IN expression COLON suite
Rule 257   for_loop -> FOR IDENTIFIER IN error COLON suite
Rule 258   for_loop -> FOR error IN expression DOTDOT expression COLON
           suite
Rule 259   for_loop -> FOR IDENTIFIER IN error DOTDOT expression COLON
           suite
```

```
Rule 260   for_loop -> FOR IDENTIFIER IN expression DOTDOT error COLON
           suite
Rule 261   conditional -> IF error COLON suite conditional_extension
Rule 262   conditional_extension -> ELIF error COLON suite
           conditional_extension
Rule 263   except_alias -> EXCEPT error AS IDENTIFIER COLON suite
           exceptlist
Rule 264   except_alias -> EXCEPT IDENTIFIER AS error COLON suite
           exceptlist
Rule 265   except_specific -> EXCEPT error COLON suite exceptlist
Rule 266   indexing -> expression LBRACKET error RBRACKET
Rule 267   slicing -> expression LBRACKET error COLON expression
           optslice RBRACKET
Rule 268   slicing -> expression LBRACKET expression COLON expression
           error RBRACKET
Rule 269   procedure_call -> expression LPAREN error RPAREN
Rule 270   function_type -> FN LPAREN error RPAREN ARROW type
Rule 271   procedure_type -> PROC LPAREN error RPAREN
Rule 272   generic_type -> IDENTIFIER LT error GT
Rule 273   dictionary_type -> DICT error LBRACKET type BITOR type
           RBRACKET
Rule 274   dictionary_type -> DICT OF LBRACKET error BITOR type
           RBRACKET
Rule 275   dictionary_type -> DICT OF LBRACKET type BITOR error
           RBRACKET
Rule 276   tuple_type -> TUPLE error LPAREN tuple_typelist RPAREN
Rule 277   tuple_type -> TUPLE OF LPAREN error RPAREN
Rule 278   function_literal -> LAMBDA LPAREN error RPAREN ARROW type
           COLON LPAREN expression RPAREN
Rule 279   function_literal -> LAMBDA LPAREN argumentlist RPAREN ARROW
           error COLON LPAREN expression RPAREN
Rule 280   function_literal -> LAMBDA LPAREN argumentlist RPAREN ARROW
           type COLON LPAREN error RPAREN
Rule 281   procedure_literal -> LAMBDA LPAREN error RPAREN COLON
           LPAREN statement_simple RPAREN
Rule 282   procedure_literal -> LAMBDA LPAREN argumentlist RPAREN
           error LPAREN statement_simple RPAREN
Rule 283   procedure_literal -> LAMBDA LPAREN argumentlist RPAREN
           COLON LPAREN error RPAREN
Rule 284   list_literal -> LBRACKET error RBRACKET
Rule 285   dictionary_literal -> LCURLY error RCURLY
```

CSPy Built-in Documentation (cspy_builtins.py)

Contains two data structures, `type_obj` and `signature`, for the typing system, as well as the built-in library (built-in types and functions).

class type_obj: a data type object class

Attributes

type_str : `string` - string representation of the data type
type : `type_obj` - type this object is an instance of
super : `type_obj` - super type of the type
elem_type : `[type_obj]` - element type for container types (lists, tuples, etc.)
sig : signature - signature for function or procedures
methods : `dict of [string | type_obj or [type_obj]]` - method dict

Methods

```
__init__(type_str:string, typ:type_obj, ?sup:type_obj = None,
        ?elem_type:[type_obj] = None, ?sig:signature = None,
        ?methods:dict of [string |type_obj or [type_obj]] = {})
```

Initializes all of the attributes for a `type_obj` to their argument values or their default values.

```
lookup_method(name:string) -> list of type_obj
```

Returns a list of class methods whose identifier is ‘`name`’ that the current type has access to either via its own method dictionary or the inherited method dictionary of its super type.

```
__eq__(other:type_obj) -> bool
```

Overrides the ‘==’ operator for `type_obj`’s. Returns true if the current `type_obj` has the same type signature as the other `type_obj`.

Note: A container type whose element type is `None`, (i.e. empty list, tuple, etc.) is equivalent to its matching container type whose element type is declared (e.g. list of int). This allows declarations and assignments such as `p:list of int = []`, or comparisons such as `if p == []:`.

```
__ne__(other:type_obj) -> bool
```

Overrides the ‘`!=`’ operator for `type_obj`’s. Returns true if the current `type_obj` does NOT have the same type signature as the other `type_obj`.

```
__repr__() -> string
```

Returns a string representation of a `type_obj` for printing purposes.

class signature: a function or procedure signature class

Attributes

`param_types : list of type_obj` - list of the parameter types
`default_types : list of type_obj` - list of the default parameter types
`return_type : type_obj` - return type (only applies to functions)

Methods

`__init__(?params:list of type_obj, ?defaults:list of type_obj, ?ret:type_obj = None)`

Initializes the attributes of a signature object to their argument values or default values.

`__eq__(other:signature)`

Overrides the '==' comparison operator for signature objects. Returns true if the current signature object has the same parameter, default, and return types as the other signature.

Note: Only used for binding a variable with function or procedure type to a function or procedure. The comparison operator is not used to check function signatures. The function `callmatch` is used instead (see type checker `signatureMatch` function).

Built-in Types

Integer: A numeric type.

Binary Operators

SYMBOL	NAME
+	add
-	subtract
*	times
/	divide
//	floor divide
**	power
%	modulo

&	bit and
|	bit or
^	bit xor
~	bit invert
==	equals
!=	not equals
>	greater than
<	less than
>=	GE than
<=	LE than
<<	left shift
>>	right shift

Unary Operators

SYMBOL	NAME
+	positive
-	negative

Type Conversion

- Float via the '`tofloat`' built-in function (see built-in function section below)
- String via the '`tostring`' or '`repr`' built-in function (see built-in function section below)

Additional Operations

Supports augmented assignment.

Float: A numeric type.

Binary Operators

SYMBOL	NAME

+	add
-	subtract
*	times
/	divide
//	floor divide
**	power
%	modulo
==	equals
!=	not equals
>	greater than
<	less than
>=	GE than
<=	LE than

Unary Operators

SYMBOL	NAME
+	positive
-	negative

Type Conversion

- Int via the '`toint`' or '`round`' built-in function (see built-in function section below)
- String via the '`tostring`' or '`repr`' built-in function (see built-in function section below)

Additional Operations

Supports augmented assignment.

Bool

Note: In Python, bool is a subclass of Integer, therefore Integer binary operators such as '+', '-', etc. would be applicable on Boolean values as well. This is currently not the case in CSPy, as we determined there would be no immediate benefit for a beginner programmer to be able to, for example, use the modulo operator on a Boolean value, or be able to add an integer to a Boolean, as this seems counterintuitive to the intended typing system. However, bool can be easily implemented in CSPy with Integer as its superclass should this become desirable later in development. It could be a possible “option” in the CSPy GUI, like importing.

Binary Operators

<table>
<tr><th>SYMBOL</th><th>NAME</th></tr>
<tr><td><code>and</code></td><td>Boolean and</td></tr>
<tr><td>&&</td><td>Boolean and</td></tr>
<tr><td><code>or</code></td><td>Boolean or</td></tr>
<tr><td><code>||</code></td><td>Boolean or</td></tr>
</table>

Unary Operators

SYMBOL	NAME
`not`	Boolean not

Type Conversion

- String via the ‘`tostring`’ or ‘`repr`’ built-in function (see built-in function section below)

String: A sequence type.

Binary Operators

SYMBOL	NAME
+	concatenate
*	repetition
>	greater than
<	less than
>=	GE than

<=	LE than
!=	equals
==	not equals

Methods

* Methods and descriptions taken from tutorialspoint.com.

`capitalize() -> string`

Capitalizes a string.

`center(width:int, ?fillchar:string = " ") -> string`

Returns a centered string of length width whose padding is done using the specified fill character. The default fill character is a space.

`count(str:string, ?beg:int = 0, ?end: int = len(string)) -> int`

Counts how many times `str` occurs in string or in a substring of the current string if the starting index `beg` and ending index `end` are given.

`decode(?encoding = "UTF-8", ?errors = "strict") -> string`

Decodes the current string using the codec encoding, which defaults to the default string encoding. Errors is the error handling scheme, which defaults to "strict", meaning encoding errors will raise a `UnicodeError`.

`encode(?encoding = "UTF-8", ?errors = "strict") -> string`

Encodes the current string using the codec encoding, which defaults to the default string encoding. Errors if the error handling scheme, which defaults to `"strict"`, meaning encoding errors will raise a `UnicodeError`.

`endswith(suffix:string, ?beg:int = 0, ?end:int = len(string)) -> bool`

Determines whether or not the current string ends with `suffix` (or a substring of a string if starting index `beg` and ending index `end` are given).

`expandtabs(?tabsize:int = 8) -> string`

Expands tabs in string to multiple spaces. Defaults to 8 spaces per tab if `tabsize` is not provided.

`find(str:string, ?beg:int = 0, ?end:int = len(string)) -> int`

Determines if `str` occurs in the current string or in a substring if starting index `beg` and ending index `end` are given. Returns starting index of `str` if found, else returns `-1`.

`index(str:string, ?beg:int = 0, ?end:int = len(string)) -> int`

Same as `find`, but raises an exception is `str` is not found.

`isalnum() -> bool`
Returns true if string has at least 1 character and all the characters are alphanumeric.

`isalpha() -> bool`
Same as `isalnum`.

`isdigit() -> bool`
Returns true if the string contains only digits.

`islower() -> bool`
Returns true if string has at least 1 cased character and all cased characters are in lowercase.

`isnumeric() -> bool`
Returns true if a Unicode string contains only numeric characters.

`isspace() -> bool`
Returns true if the string contains only whitespace characters.

`istitle() -> bool`
Returns true if the string is properly "titlecased".

`isupper() -> bool`
Returns true if string has at least one cased character and all cased characters are in uppercase.

`join(seq:list/tuple of string) -> string`
Concatenates the elements in the sequence into a string with the current string as a separator. The elements in the sequence must be strings.

`ljust(width:int, ?fillchar:string = " ") -> string`
Returns a left justified string of length `width` whose padding is `fillchar`, which defaults to a space.

`lower() -> string`
Converts all uppercase letters to lowercase.

`lstrip() -> string`
Removes all leading whitespace in the current string.

`replace(old:string, new:string, max:int) -> string`
Returns a copy of the string with *all* occurrences of the substring `old` replaced by `new` if `max` is not specified. If `max` is specified, only `max` occurrences will be replaced starting from the front of the string.

rfind(str:string, ?beg:int = 0, ?end:int = len(string)) -> int
Same as `find()`, but searches backwards in the string.

rindex(str:string, ?beg:int = 0, ?end:int = len(string)) -> int
Same as `index()`, but searches backwards in the string.

rjust(width:int, ?fillchar:string = " ") -> string
Returns the original string right justified to a total width of columns using `fillchar`, which defaults to a space.

rstrip() -> string
Removes all of the trailing whitespace on a string.

split(str:string = " ", ?num:int = string.count(str)) -> list of string
Splits strings according to `str` (defaults to a space) and returns a list of substrings. Splits into at most `num` substrings if `num` is given.

splitlines(?num:int = string.count("\n")) -> list of string
Splits at all (or `num` if given) new lines and returns a list of each line with newlines removed.

startswith(str:string, ?beg:int = 0, ?end:int = len(string)) -> bool
Determines if the current string (or a substring of string if the starting index `beg` and ending index `end` are given) starts with the substring `str`.

strip() -> string
Performs both `lstrip()` and `rstrip()` at the same time.

swapcase() -> string
Inverts the case for all letters in a string.

title() -> string
Returns "titlecased" version of the current string where all words begin with uppercase letters and the rest are lowercase.

upper() -> string
Converts lowercase letters in the current string to uppercase.

Type Conversion

- Integer via '`toint`' built-in function (see built-in function section below)
- Float via '`tofloat`' built-in function
- list, set, or frozenset via '`to`*`type`*' built-in functions

Additional Operations

String supports indexing, slicing, membership, iterations, and use of the `len()` function. Also supports augmented assignment.

List: A sequence type.

Lists are homogenous, meaning they can only contain elements of the same type. Lists are mutable.

Binary Operators

SYMBOL	NAME
+	concatenate
*	repetition
!=	not equals
==	equals

Methods

* Methods and descriptions taken from tutorialspoint.com.

Note: The function signatures of list methods are specific to their element type, which is why the method dictionary for a list is constructed in the function ‘`init_list`’. This prevents things like `mylist.add(“a”)` where `mylist` is a `list of int`.

`append(obj:elem_type)`
Appends `obj` to the end of the list.

`count(obj:elem_type) -> int`
Returns count of how many times `obj` occurs in list.

`extend(seq:list of elem_type)`
Appends the contents of `seq` to list.

`index(obj:elem_type) -> index`
Returns the first index in list where `obj` appears.

`insert(index:int, obj:elem_type)`
Inserts `obj` into list at offset `index`.

`pop(?index:int = -1) -> elem_type`

Removes and returns the object at `index` from list, or the end of the list if `index` was not given.

`remove(obj:elem_type)`
Removes `obj` from list.

`reverse()`
Reverses the order of the objects in the list.

`sort()`
Sorts the objects in the list.

Type Conversion

- string via '`tostring`' or '`repr`' built-in function (see built-in function documentation)
- set or frozenset via '`makeset`' or '`frzset`' built-in functions
- bool via '`tobool`' built-in function

Additional Operations

Lists support indexing, slicing, membership testing, iteration, and use of the `len()` function. Also supports augmented assignment.

Tuple: A sequence type.

Tuples are heterogeneous, i.e. can contain elements of multiple types. Tuple are immutable - they cannot be modified once created.

Binary Operators

SYMBOL	NAME
+	concatenate
*	repetition
!=	not equals
==	equals

Type Conversion

- list, set, frozenset type via '`tolist`', '`makeset`', and '`frzset`' built-in functions (only applicable for homogeneous tuples)
- string via '`tostring`' or '`repr`' built-in function (see built-in function documentation)

Additional Operations

Tuples support indexing, slicing, membership testing, iteration, and use of the `len()` function. Tuples do not support augmented assignment (they are immutable).

Note: The methods for tuples have generic "object" return types because the return type of these operators depends on the type of the tuple. Since tuples can contain multiple types unlike lists, the return type of a slicing or indexing operation is determined in the type checker by analyzing the `elem_type` list of the tuple. Because it is impossible to determine the return type of an indexing or slicing operation of a multi-typed tuple using a variable as an index, indexing tuples with variables is NOT allowed.

`mytuple[2]` → acceptable
`mytuple[p]` → unacceptable

Similarly, because it is impossible to determine the type of an iterative variable in a for loop iterating over a multi-type tuple, tuples DO NOT support iterating.

Dictionary: An associative array.

The keys and values for a dictionary are homogenous: all of the keys must have the same type, and all of the values must have the same type. The key type and the value type may be different, e.g. integer keys with string values. Dictionaries are mutable.

Binary Operators

SYMBOL	NAME
!=	not equals
==	equals

Methods

* Methods and descriptions taken from tutorialspoint.com.

Note: Similar to lists, the type signature of dictionary method depends on the type of the dictionary instance's key type and value type, which is why there is an '`init_dict`' function instead of a singular built-in method dictionary for the dictionary type.

`clear()`
Removes all elements of the current dictionary.

`copy() -> dict of [key_type|value_type]`
Returns a shallow copy of the current dictionary.

get(key:key_type, default:value_type) -> value_type

For key, returns its value, or default if key not in dictionary.

Note: In Python, 'default' defaults to 'None' type, which currently does not exist in CSPy. Therefore 'default' is required in CSPy.

has_key(key:key_type) -> bool

Returns true if key is in dictionary.

*items() -> list of tuple of (key_type * value_type)*

Returns a list of the current dictionary's (key, value) tuple pairs.

keys() -> list of key_type

Returns list of the current dictionary's keys.

pop(elem:key_type, ?default:value_type) -> value_type

If elem is in the current dictionary, removes elem from the dictionary and returns its value. If elem is not in the current dictionary and default was not given, raises KeyError.

popitem() -> tuple of (key_type * value_type)

Removes and returns an arbitrary (key, value) pair from the current dictionary. If the dictionary is empty, calling popitem() raises a KeyError.

setdefault(key:key_type, v:value_type)

Sets dict[key] = v if key is not already in the current dictionary.

Note: In Python, 'v' defaults to 'None' type, which currently does not exist in CSPy. Therefore 'v' is required in CSPy.

update(dict2:dict of [key_type|value_type])

Adds dictionary dict2's key-value pairs to the current dictionary.

values() -> list of value_type

Returns a list of the values in the current dictionary.

Type Conversion

- string via 'tostring' or 'repr' built-in function (see built-in function documentation)
- list, set, frozenset via 'tolist', 'makeset', and 'frzset' built-in functions
- bool via 'tobool' built-in function

Additional Operations

Dictionaries support indexing, slicing, membership testing, iteration, and use of the `len()` function.

Set and Frozenset

An unordered collection of unique elements. Sets and frozensets are homogeneous, i.e. they can only contain one element type. Sets are mutable but frozensets are immutable.

Binary Operators

SYMBOL	NAME
<	proper subset
>	proper superset
<=	subset
>=	superset
|	union
&	intersection
-	difference
^	symmetric difference
!=	not equal
==	equals

Methods

* Methods and descriptions taken from the Python documentation.

Note: The function signatures of set and frozenset methods depend on the element type, like lists and dictionaries. There are ‘`init_set`’ and ‘`init_frzset`’ functions that generate a typed method dictionary for their respective types.

`isdisjoint(s:set/frozenset of elem_type) -> bool`

Returns true if the current set is disjoint from s (the set has no elements in common with s).

`issubset(s:set/frozenset of elem_type) -> bool`

Returns true if the current set is a subset of s (every element of the current set is in s).

issuperset(s:set/frozenset of elem_type) -> bool

Returns true if the current set is a superset of s (every element of s is in the current set).

union(s:set/frozenset of elem_type) -> set/frozenset of elem_type

Returns a new set that is the union of the current set and s (a set containing all elements from current set and s).

intersection(s:set/frozenset of elem_type) -> set/frozenset of elem_type

Returns a new set that is the intersection of the current set and s (a set with all elements that are in both the current set and s).

difference(s:set/frozenset of elem_type) -> set/frozenset of elem_type

Returns a new set with all elements in the current set that are not in s.

symmetric_difference(s:set/frozenset of elem_type) -> set/frozenset of elem_type

Returns a new set with all elements in either the current set or s but not both.

copy() -> set/frozenset of elem_type

Returns a shallow copy of the current set.

Set ONLY methods - do not apply to frozensets

update(s:set)

Update the current set by adding all elements from set s.

intersection_update(s:set):

Update the current set by keeping only elements found in both the current set and the set s.

difference_update(s:set):

Update the current set by keeping only elements found in either the current set or s, but not in both.

add(elem:elem_type)

Add the element `elem` to the current set.

remove(elem:elem_type)

Removes element `elem` from the current set. Raises `KeyError` if `elem` is not in the current set.

`discard(elem:elem_type)`

Same as `remove` but does not raise `KeyError` if `elem` is not present in the current set.

`pop() -> elem_type`

Removes and returns an arbitrary element from the current set. Raises `KeyError` if the set is empty.

`clear()`

Removes all elements from the current set.

Type Conversion

- string via '`tostring`' or '`repr`' built-in function (see built-in function documentation)
- list, set, frozenset via '`tolist`', '`makeset`', '`frzset`' built-in functions
- bool via '`tobool`' built-in function

Additional Operations

Sets and frozensets support membership testing and iteration. Sets support augmented assignment. Frozensets do not (they are immutable).

Function

A function is a procedure that returns a value. Functions have a return type. All of the return statements in a function must be nonempty and their return value must have the same type as the function's return type.

CSPy supports function overloading, meaning multiple function definitions can have the same identifier, provided that each one has a unique parameter type list. The return types do not factor into overloading. Consider the following function definitions for an example of unacceptable overloading:

```
def add(x:int, y:int) -> int:
   return x + y

def add(x:int, y:int) -> float:
   return tofloat(x + y)
```

These function definitions for `add` have the same parameter type lists (int, int) and so this overloading is not allowed. This is because there is no way for the type checker (or the translator, for that matter) to determine which `add` is being referred to when `add(1,4)` is called. Attempting an overload in this manner will result in an error from the type checker.

```
def add(x:int, y:int) -> int:
```

```
    return x + y

def add(x:float, y:float) -> float:
    return x + y
```

The definitions of `add` above are an example of valid function overloading, because the two functions have different parameter type lists and are therefore distinguishable.

A declared variable may be bound to a function:

```
f:fn (int, int) -> int = add
```

The above will assign the overloaded `add` function corresponding to the given type signature to `f`. Similarly, the below example is also valid because the type of the anonymous function matches the type of `f`:

```
f:fn (int, int) -> int = lambda (x:int, y:int) -> int : (x + y)
```

Note: `fn (int, ?int) -> int` is not equivalent to `fn (int, int) -> int`. The '?' symbol in the first function type indicates the second integer is an optional parameter. This is the same for procedures.

Procedure

A procedure does not return a value and hence has no return type. Procedures support function overloading as well. An example of valid procedure overloading is below:

```
def output(i:int):
    print(i)

def output(f:float):
    print(f)
```

Declared variables may be bound to procedures using the following syntax:

```
p:proc (int) = output
```

File

File is the type for a Python file object. A file is created by using the built-in `open()` function.

Attributes

`closed:bool` - True if the file is closed, False otherwise
`name:string` - The name of the file
`mode:string` - The mode which the file was opened with

Methods

* methods and descriptions taken from Python documentation

`close()`

Closes the file. A closed file cannot be read or written to anymore. Any operation which requires that the file be open will raise a `ValueError` is the file is closed. Calling *`close`* more than once is allowed.

`flush()`

Flushes the internal buffer.

`fileno() -> int`

Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.

`next() -> string`

Returns the next line from the file each time it is being called.

`read(?size:int = file size) -> string`

Read at most `size` bytes from the current file, less if hits `EOF` before reaching `size` bytes. If `size` is not given, reads the entire file.

`readline(?size:int = file size) -> string`

Reads one line from the file. If the `size` argument is present, it is a maximum byte count of the line. An empty string is returned only when `EOF` is encountered immediately.

`readlines(?size:int = file size) -> list of strings`

Reads until `EOF` using `readline` and return a list containing the lines. If `size` is given, instead of reading up to `EOF`, reads whole lines totaling approximately `size` bytes in size.

`seek(offset:int, ?whence:int = 0)`

Sets the current file position to `offset`. If `whence` is given, sets the current position to the `offset` from `whence`.

`tell() -> int`

Returns the file's current position.

`truncate(?size:int = ?)`

Truncates the file size. If `size` is given, the file is truncated to at most that size.

`write(str:string)`

Writes `str` to the current file.

writelines(seq:list of string)

Writes a sequence of strings from a list to the current file.

Built-in Functions

* methods and descriptions taken from Python's documentation.

Note: Not all of the built-in Python functions are currently implemented in CSPy. The below functions have been implemented.

abs(x:int) -> int
abs(x:float) -> float

Returns the absolute value of `x`.

all(l:list of ?) -> bool

Returns true if all the elements in `l` are true.

any(l:list of ?) -> bool

Returns true if any of the elements in `l` are true.

bin(x:int) -> string

Converts `x` into a binary string.

chr(i:int) -> string

Returns a string representing a character whose Unicode point is `i`.

cmp(a:int, b:int) -> int
cmp(a:float, b:float) -> int
cmp(a:string, b:string) -> int
cmp(a:bool, b:bool) -> int
cmp(a:list, b:list) -> int
cmp(a:tuple, b:tuple) -> int
cmp(a:dict, b:dict) -> int
cmp(a:set, b:set) -> int
cmp(a:frozenset, b:frozenset) -> int

Returns `1` if *`a > b`*, `-1` if *`a < b`*, and `0` if *`a == b`*.

*divmod(a:int, b:int) -> tuple of (int * int)*
*divmod(a:float, b:float) -> tuple of (float * float)*

Returns a pair of numbers consisting of the quotient of `a` and `b` and their remainder when using integer division. For integers, this is equivalent to (`a // b`, `a % b`). For floats, this is equivalent to (`math.floor(a / b)`, `a % b`).

exit(?code:int)

Exits from the current program.

hex(x:int) -> string

Converts x to a lowercase hexadecimal string prefixed with '0x'.

```
len(s:string) -> int
len(l:list) -> int
len(t:tuple) -> int
len(d:dict) -> int
len(mset:set) -> int
len(fr:frozenset) -> int
```

Returns the number of objects in the given sequence or container.

```
max(a:int, b:int) -> int
max(a:float, b:float) -> float
max(l:list of elem_type) -> elem_type
```

For integers and floats, returns a if a > b or b if b > a. For lists, returns the item from the list with max value.

```
min(a:int, b:int) -> int
min(a:float, b:float) -> float
min(l:list of elem_type) -> elem_type
```

For integers and floats, returns a if a < b or b if b < a. For lists, returns the item from the list with min value.

```
oct(x:int) -> string
```

Converts x to an octal string.

```
ord(s:string) -> int
```

Given a string representing a Unicode character s, returns an integer representing the Unicode point of s.

```
open(name:string, mode:string = "r") -> file
```

Open the file name in mode. If mode is not given, defaults to “r” (read). Returns a file object.

```
pow(x:int, y:int) ->
pow(x:int, y:int, z:int) -> int
```

Returns x to the power y. If z is present, returns x to the power y modulo z.

```
range(stop:int) -> list of int
range(start:int, stop:int, step:int = 1 ) - > list of int
```

Returns a list of integers representing the range of integers from start to stop using step if given. If only stop is given, start defaults to 0.

```
round(x:float, y:int = 0) -> int
```

Returns x rounded to y digits after the decimal point. If y is omitted, returns the nearest integer to its input.

sum(l:list of int, start:int = 0) -> int
sum(l:list of float, start:float = 0.0) -> float
sum(t:tuple of int, start:int = 0) -> int
sum(t:tuple of float, start:float = 0.0) -> float
sum(t:set of int, start:int = 0) -> int
sum(t:set of float, start:float = 0.0) -> float
sum(t:frozenset of int, start:int = 0) -> int
sum(t:frozenset of float, start:float = 0.0) -> float

Sums `start` and the items of the iterable from left to right and returns the total. `start` defaults to `0` if not given.

Type Checking Functions (cspy_type_checker.py)

These functions

`s_argumentlist(n:ast)`

Assigns the type of `n`, an argument list node, to a signature object with the appropriate parameter and default types.

`s_argumentlist_single(n:ast)`

Assigns the type of `n`, a single argument list node, to the type of its child node.

`s_assignment(n:ast)`

There are two types of assignment: normal assignment using the "=" operator, or augmented assignment using a binary operator. For normal assignment, ensures the variable on the LHS of the assignment has the same type of the expression it is being bound to.

e.g. `x = "Hi"` where `x` is an integer will result in a type error.

For augmented assignment (e.g. "+="), calls the function `binaryop` on the current node, which checks if both the LHS and RHS support use of the binary operator used in the augmented assignment and ensures their signatures match.

e.g. `x += 4.0` where `x` is an integer will result in a type error

`s_calculation_binaryoperator(n:ast)`

Calls `binaryop` on `n`, which type errors if the binary operator is undefined for either the LHS or the RHS, or if the signature of the binary operator for the LHS does not match the RHS or vice versa.

`s_calculation_unaryoperator(n:ast)`

Checks if the operand has the unary operator defined. Type errors if the unary operator is undefined for the operand, or if it is defined, but is not a function (only relevant if a user defined class overrides the operator).

`s_conditional(n:ast)`

Type errors if the condition of an if/elif statement is not a Boolean expression.

Note: Python's truth testing allows for an object to be tested for a truth value.

`s_declaration_initalization(n:ast)`

For a variable declaration with an initialization step, checks if the given value has the same type as the variable's declared type. Type errors if the types do not match.

e.g. `x:int = "Hello World"` will result in a type error

`s_defaultlist_single(n:ast)`

Type errors if a parameter's default value does not match its declared type.

e.g. `def pow(x:int, y:int = "10")` will result in a type error *s_default_typelist_single(n:ast)*

Sets the type of a single default type list node to the types of its child node (i.e. type of the type literal).

s_dictionary_type(n:ast):

Sets the type of a dictionary type node to a dictionary type object whose method dictionary corresponds to the key type and the value type specified in the type declaration.

s_except_specific(n:ast)

Example: for '`except ValueError`', type errors if `ValueError` is undefined, or if `ValueError` is defined but is not an exception type.

s_except_alias(n:ast)

Example: for '`except ValueError as v`', type errors if `ValueError` is undefined, or if `ValueError` is defined but is not an exception type.

s_expressionlist_single(n:ast)

Assigns the type of a single expression list node to the type of its child node.

s_forloop_iter(n:ast)

Type errors if the object being iterated over is not a sequence or file type.

e.g. `for i in x` where `x` is an integer will result in a type error

s_forloop_count(n:ast)

Type errors if the start and stop values of a dotdot range are not integers. (Note: This type of for loop is essentially a for loop using `range()`, but with an inclusive stop value and no step. e.g. `for x in 1..4` equals `for x in range(1,5)`)

e.g. `for i in 2.0..4.0` will result in a type error

s_frozenset_type(n:ast)

Sets the type of a frozenset type node to a frozenset type object whose element type and method dictionary correspond to the given type in the declaration.

s_function_type(n:ast)

Assigns the type of a function type literal (e.g. `fn (int) -> int`) to a function type object with the appropriate signature.

s_function_definition(n:ast)

Type errors if a function definition contains a return statement whose type does not match the function's specified return type. Type errors if the function definition contains empty return statements. Type errors if the function is missing a return statement.

Note: Checking for non-exhaustive return statements is currently unimplemented.

s_function_call(n:ast)

Determines the signature of a function call from its parameter types. Type errors if the identifier does not correspond to a function. If the identifier has multiple values (function overloading has occurred), `signatureMatch` determines whether or not any of the signatures of the overloaded function correspond to the signature of the function call; type errors if there is no signature for the identifier matching the function call.

Constructors: A constructor call is classified by the parser as a function call. Constructors do not return anything, therefore a constructor definition is classified by the parser as a procedure definition. Whenever the type checker encounters a procedure call, it will change the label on the node (for the translator's ease of use) and call `s_procedure_call` on `n`.

Built-in functions: There are specific built-functions (like type conversions to container types) which require special type checking, or whose signature depends on the types of their arguments. In `cspy_builtins.py`, there are numerous built-in functions whose dictionary value corresponds to a Python function, also defined in `cspy_builtins.py`. These functions perform any special type checking CSPy requires, but native Python does not (e.g. a non-homogeneous tuple cannot be converted to a list).

s_grouping(n:ast)

Assigns the type of a grouping node to the type of its child expression node.

s_indexing(n:ast)

Looks up the '`__getitem__`' method for the object being indexed. Type errors if the method is undefined. If the method is defined, checks the signature of the method against the type of the index. Type errors if the index has the wrong type. Assigns the type of n to the return type of the '`__getitem__`' method.

e.g. `x[0]` where `x` is an integer will result in a type error

s_list_type(n:ast)

Sets the type of a list type node to a list type object whose element type and method dictionary correspond to the given type in the declaration.

s_literal_bool(n:ast)

Assigns bool type object to a bool literal node.

s_literal_dictionary(n:ast)

Assigns dictionary type object to a dictionary literal node if the dictionary keys are homogeneous and the dictionary values are homogeneous, else type errors.

e.g. `{1 : “a”, “2” : “b”}` will cause a type error

s_literal_float(n:ast)

Assigns float type object to a float literal node.

`s_literal_function(n:ast)`

Checks the types of a function literal (i.e. a lambda expression). Type errors if the return type of the function does match the type of its body. Sets the type of n to a function type object with the appropriate function signature.

`s_literal_int(n:ast)`

Assigns int type object to an int literal node.

`s_literal_list(n:ast)`

Assigns list type object to a list literal node if the list elements are homogeneous, else type errors.

e.g. `[1, 2, "a"]` will result in a type error

`s_literal_set(n:ast)`

Assigns set type object to a set literal node if the set elements are homogenous, else type errors.

e.g. `{1, 2, "3"}` will result in a type error

Note: `{}` is not an empty set literal, it is an empty dictionary literal.

`s_literal_string(n:ast)`

Assigns string type object to a string literal node.

`s_literal_tuple(n:ast)`

Assigns tuple type object whose element types consist of the type of the values in the tuple to a tuple literal node.

`s_member(n:ast)`

syntax : `object.attribute`

Looks up attribute in the method/attribute dictionary of object. Type errors if the lookup fails (the object or its object's supertype does not have the method/attribute).

e.g. `mytuple.append(4)` will result in a type error (tuples are immutable)

`s_membership(n:ast)`

Checks whether or not a type supports membership testing. Only sequence types support membership testing. Type errors for membership testing on non-sequence types.

e.g. `if 4 in 4`: will result in a type error

Type errors for membership testing of a value whose type does not match an element type of the sequence. An alternative to this particular kind of type error (e.g. `if 1.0 in [1,2,3]`) is a warning which allows the program to run, as opposed to an error message – this is not yet implemented, though ideal.

e.g. “WARNING: The given conditional statement will always evaluate as False because the object type does not match the container type, and so the object can never be a member of the container.”

`s_procedure_type(n:ast)`

Assigns the type of a procedure type literal node (e.g. `proc (string)`) to a procedure type object with the appropriate signature.

`s_procedure_defintion(n:ast)`

Type errors if a procedure contains a non-empty return statement.

Note: Checking for nonexhaustive returns statements currently not implemented.

`s_procedure_call(n:ast)`

Determines the signature of a procedure call from its parameter types. Type errors if the identifier does not correspond to a procedure. If the identifier has multiple values (overloading has occurred), `signatureMatch` determines whether or not any of the overloaded procedure signatures correspond to the signature of the procedure call; type errors if there is no signature for the identifier matching the procedure call.

Built-in functions: There are specific built-functions (like type conversions to container types) which require special type checking, or whose signature depends on the types of their arguments. In `cspy_builtins.py`, there are numerous built-in functions whose dictionary value corresponds to a Python function, also defined in `cspy_builtins.py`. These functions perform any special type checking CSPy requires, but native Python does not (e.g. a non-homogeneous tuple cannot be converted to a list).

`s_return(n:ast)`

Sets the type of a return statement node to the type of its child node (the type of the value being returned, if there is one).

`s_set_type(n:ast)`

Sets the type of set type node to a set type object whose element type and method dictionary correspond to the given type in the set declaration.

`s_slicing(n:ast)`

Looks up the ‘`__getslice__`’ method for the object being sliced. Type errors if the method is undefined. If the method is defined, checks the signature of the method against the type of the start, stop, and step indices. Type errors if any of the indices have the wrong type. Assigns the type of `n` to the return type of the ‘`__getslice__`’ method.

e.g. `x[1:]` where `x` is an integer will result in a type error;
`l[“1”:]` where `l` is a list will result in a type error

`s_ternary(n:ast)`

Checks to see if the condition of the ternary operator is a Boolean expression. Type errors if the if-then expression does not have the type of the else expression. Assigns the type of `n` to a type object whose type is the type of the resultant if-then/else expression.

e.g. `(x == 1) ? True : “False”` will result in a type error

`s_tuple_type(n:ast)`

Sets the type of a tuple type node to a tuple type object whose element type corresponds to the given types in the tuple declaration.

`s_tuple_typelist_single(n:ast)`

Sets the type of a single tuple type list node to the type of its child node.

`s_type(n:ast)`

For an identifier (e.g. int, bool, float), looks up its type. Type errors if the identifier is undeclared (e.g. `x:integer = 4`), i.e. has no value (“`integer`” is not a defined variable). Type errors if the identifier is declared but it is not a type (e.g. `x:y` where `y` is an integer). Otherwise, assigns the type of the node to the type of the identifier.

`s_typelist(n:ast)`

Creates a signature object from a list of types from the parameters of a function type literal or a procedure type literal (e.g. `fn` **`(int, string)`** `-> int`)

Note: A type identifier preceded by a ‘?’ symbol indicates the type corresponds to a parameter with a default value (e.g. `fn` **`(int, ?string)`** `-> int` is a function type corresponding to a function whose second argument is a string with a default value.

`s_typelist_single(n:ast)`

Sets the type of a single type list node to the type of its child (i.e. the type of the type literal).

`s_variable(n:ast)`

Looks up the type of the variable within its scope. Type errors if the variable is undeclared; else, sets the type of `n` to the type of the variable.

`s_whileloop(n:ast)`

Type errors if the condition of a while loop is not a Boolean expression.

Note: Python’s truth testing allows any object to be tested for truth value.