CSPy: A Python Dialect

Lyndsay LaBarge ‘17, Maya Montgomery ‘18, and Alistair Campbell, Associate Professor of Computer Science

Clare Boothe Luce Undergraduate Research Scholarship Recipients

Hamilton

What is CSPy?

At Hamilton, introductory Computer Science students are taught how to program using Python. But Python does not necessarily teach students good coding habits, partially due to its dynamic typing system. Type errors can cause bugs for beginner programmers that are difficult to trace back. Designed by Hamilton students for novice programmers, CSPy is a statically typed dialect of Python with explicitly declared and typed variables.

CSPy has been in development for three years. Work up until this point has included language design and the production of a robust grammar and parser, as well as basic semantic analysis. By expanding the foundation built by previous students, we were able to develop a comprehensive type-checker and system for translating CSPy source code to native Python to be run by the Python interpreter.

Features of CSPy

Figure 1.1 A class definition in CSPy.

```
'''example.cspy'''

class Pet:
    :: name:string, hungry:bool = True ::

    def Pet(n:string):
        name = n

    def feed():
        hungry = False
        print(name + " is not hungry!")

    def checkHunger() -> bool:
        return hungry

def main():
    :: myDog : Pet = Pet("Spot") ::

    if myDog.checkHunger() == True:
        myDog.feed()

main()
```

Figure 1.2 A class definition in Python.

```
'''example.py'''

class Pet:

    def __init__(self, n):
        self.name = n
        self.hungry = True

    def feed(self):
        self.hungry = False
        print(self.name + " is not hungry!")

    def checkHunger(self):
        return self.hungry

def main():
    myDog = Pet("Spot")

    if myDog.checkHunger() == True:
        myDog.feed()

main()
```

CSPy has four built-in data types: int, float, bool, and string. The language also inherits Python’s built-in data structures – list, tuple, dictionary, set, and frozenset – all of which are homogeneous, with the exception of tuples. Function types are divided into two categories, functions and procedures. Procedures do not return a value and are equivalent to the function type “`void`” in other statically typed languages such as C++ and Java.

All variable declarations take place in a variable block (denoted by starting and ending ‘`::`’) at the top of a code block. The variable block below contains example declarations for CSPy’s main built-in types.

```
:: a : int,  b : float,  c : string,  d : bool,  e : list of int,
   f : tuple of (int * bool), g : set of string, h : dict of [int|string] ::
```

Function overloading is a feature of CSPy not native to Python. Functions and procedures as well as class constructors are allowed to have multiple definitions, provided each definition has a unique signature. During translation, each overloaded function definition is translated to a separate function whose identifier corresponds to its signature (e.g. `_add_int_int`). When an overloaded function is called, the translator determines the correct function name by the call signature.

CSPy eliminates the “`self`” prefix in front of class attributes and methods. Programmers must instead declare class attributes in a variable block at the top of a class definition. Additionally, programmers are no longer required to override the “`__init__`” constructor for user defined classes. Instead, the identifiers of constructors in CSPy have the same identifier as their class, and the translator writes an `__init__` in which to declare the class attributes. Thus, to instantiate a class object, the translator will output code to call the `__init__` constructor and then call the user-defined constructor as a method.

What’s Next?

There is currently no system in place to import Python modules into CSPy files. Future developers will need to develop ways to integrate Python module environments during CSPy compilation and perform type-checking operations on variables and functions from the imported Python module.

We also would like to create an IDE where programmers can write CSPy programs and run them using a single application. The IDE would include options to control the programming environment for beginners, such as the ability to turn importing on or off, or whether or not to run standalone code.

Figure 2.1

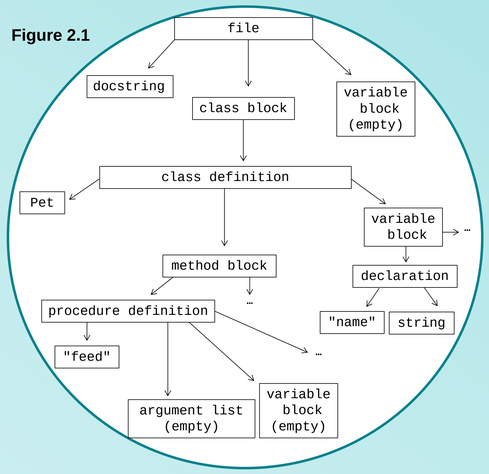

Figure 2.2

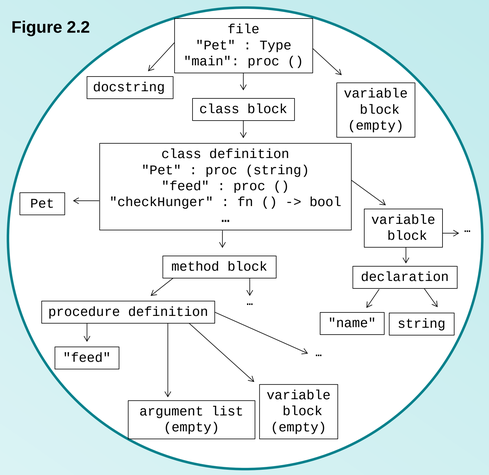

Figure 2.3

```
class Pet:
    def __init__(self):
        self.name = ""
        self.hungry = True

    def Pet(self, n):
        self.name = n

    def feed(self, ):
        self.hungry = False
        print(self.name + " is not hungry!")

    def checkHunger(self, ):
        return self.hungry

def main():
    myDog = Pet("Spot")
    myDog.Pet("Spot")

    if (myDog.checkHunger() == True):
        myDog.feed()

main()
```

Figure 2.4

```
[username ~]$ cspy example.cspy
Spot is not hungry!
[username ~]$
```

How CSPy Works

The diagram to the left follows the CSPy program in Figure 1.1 through the compilation process.

Lexical Analysis & Parsing
CSPy’s lexer and parser are implemented using PLY (Python-Lex-Yacc), a Python implementation of the popular UNIX tools lex and yacc by David Beazley. The lexing file contains a series of regular expressions corresponding to all of the tokens in the CSPy language. The lexer receives source code as input and produces a token stream; if it encounters any unknown tokens, an error will occur.

The parsing file contains function definitions, each prefixed with “`p_`” followed by the name of a nonterminal, which define the rules in the grammar for the given nonterminal. It also contains specific error rules containing a special `error` token. When the parser encounters an error, all of the tokens following the parse error will match the `error` token until the parser encounters the next valid token in a rule, at which point it will recover and continue parsing after displaying a syntax error message. If there are no syntax errors in a file, the parser will produce an abstract syntax tree.

Generating Environments
Certain nodes in the abstract syntax tree contain environments. For example, the root of the tree contains the global environment of a parsed CSPy file. To make importing CSPy modules possible, abstract syntax trees are searched for import statements. Any import modules are parsed, checked for type errors, and translated to Python. Afterwards their global environments are combined with the global environment of their parent in a way reflective of the type of import statement. Other nodes, like those corresponding to function definitions, contain environments as well. Before semantic analysis occurs, any variable declarations are added to the correct environment in the parse tree. Any class definitions are also added to the file’s global environment.

Semantic Analysis
Once a file’s environments have been created, the type checking process begins. Each node in the abstract syntax tree has a label attribute corresponding to the type of statement or expression the node contains (eg. `binary operation`). The syntax tree is traversed by mapping the label of a node to a function. Depending on the node, the function either assigns a type (such as `int`) to the node, or verifies the type-safety of the information contained within the node. See Figure 3.1 for an example of a type error.

```
[username ~]$ cspy bad_type.cspy
CSPy : Type Error
Line 3, Column 8: int
:: Z : int = "Hello!" ::
       ^^^
Line 3, Column 14: string
:: Z : int = "Hello!" ::
             ^^^^^^^^
The value assigned to 'Z' must have type 'int', not type 'string'.
[username ~]$
```

Figure 3.1

In Figure 3.1, a user attempts to run their file, "`bad_type.cspy`", but a line in the file attempts to declare variable `Z` with type `int` and a value with type `string`. This throws a type error.

Translation
After a program has been type checked, the abstract syntax tree is translated to native Python code (Figure 2.3), as it is much more efficient to translate CSPy and then use a Python interpreter than it is to write an interpreter just for CSPy. The translator traverses the AST and writes a Python translation to a new file. Translation is delegated to node label-specific functions that break down the AST into pieces small enough to interpret. With every new line output, a new entry is added to a dictionary used to map Python translation line numbers to CSPy source code line numbers. This dictionary is ultimately written to a file for use in runtime.

Execution
After translation, the resulting Python program is run by a Python interpreter as a separate process, receiving any input and returning any output into the user’s terminal (Figure 2.4). Any runtime errors are piped into a text file, which is read and rewritten into a more easily understandable format for beginners. CSPy source code line numbers are substituted for the Python executable line numbers by importing the dictionary written by the translator. See Figure 3.2 for a standard CSPy error message.

```
[username ~]$ cspy run_error.cspy

THERE IS AN ERROR IN FILE 'run_error.cspy', LINE 6:
    y = x / 0

ZeroDivisionError: integer division or modulo by zero

TRACEBACK:
File "run_error.cspy", line 8:
    divide(9)
File "run_error.cspy", line 6:
    y = x / 0
[username ~]$
```

Figure 3.2

In Figure 3.2, a user attempts to run their file, "`run_error.cspy`", but a line in the file attempts to divide by 0. This error is caught and reported during runtime. CSPy error reporting aims to make the error message more readable and to prioritize the error over the traceback, as opposed to Python error reporting.