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, students typically first learn to program using the language Python, as it is generally more intuitive than other languages. However, Python may not be the best at teaching students good coding habits, in part due to its dynamic typing system. CSPy is intended to replace Python as Hamilton’s beginner language. Designed and written by Hamilton students, CSPy is a type-safe programming language based on Python. It aims to encourage conscientious coding in introductory students. For example, CSPy requires explicitly declared and typed variables, and disallows automatic type conversions. The language attempts to create an environment where users fully understand their code.

In the summer of 2016, Lyndsay LaBarge ‘17 and Maya Montgomery ‘18 developed a functional version of CSPy; through expanding existing code and writing much code from scratch, CSPy now boasts parsing, type-checking, translation, and execution capabilities. In other words, we are now capable of writing and running CSPy programs.

CSPy vs. Python

Figure 1.1

```
'''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

```
'''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 is a statically typed dialect of Python. Python itself is dynamically typed, meaning the type of a variable is interpreted at runtime, and type errors cause running programs to terminate. Variables in Python are defined, not declared, as opposed to CSPy, where each variable must be declared with a named type. Because the type of a variable is known at compile time, CSPy source code can be checked for type errors before a program is run to reduce the amount of debugging. All declarations in CSPy take place in a variable block (denoted by starting and ending “`::`” symbols) at the beginning of a code block. The variable block below contains 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] ::
```

Figure 1.1 is an example of a class definition in CSPy; Figure 1.2 is the corresponding Python program. In Figure 1.1, the variable `myDog` is declared, assigned type “Pet”, and bound to a new `Pet` object (`myDog : Pet = Pet(“Spot”)`). Only “Pet” objects can be bound to `myDog` in CSPy. This is not the case in Python, where `myDog` is only defined and can be assigned a value of any type.

CSPy eliminates the “`self`” prefix in front of class attributes and methods, a common source of confusion for beginner programmers. Instead, attributes are declared in a variable block at the top of a class definition. Additionally, the “`__init__`” syntax for a class constructor in Python has been replaced with a procedure named after the class. Python restricts programmers to a single constructor for a class; however, CSPy allows the definition of multiple constructors for an individual class provided each method has a unique signature. This feature, called function overloading, is also supported for functions and procedures, in order to give users more flexibility in their functions.

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

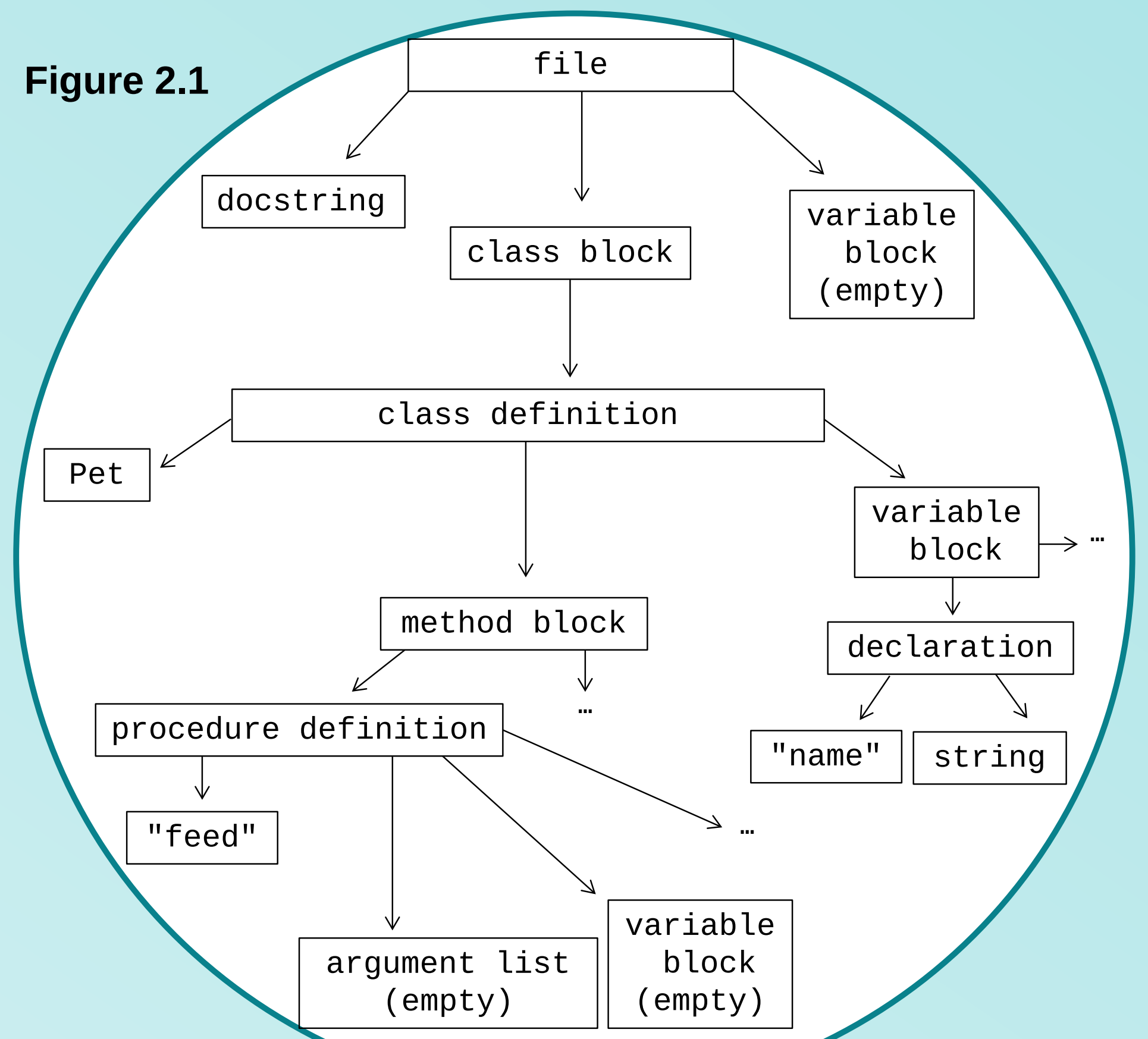

Figure 2.2

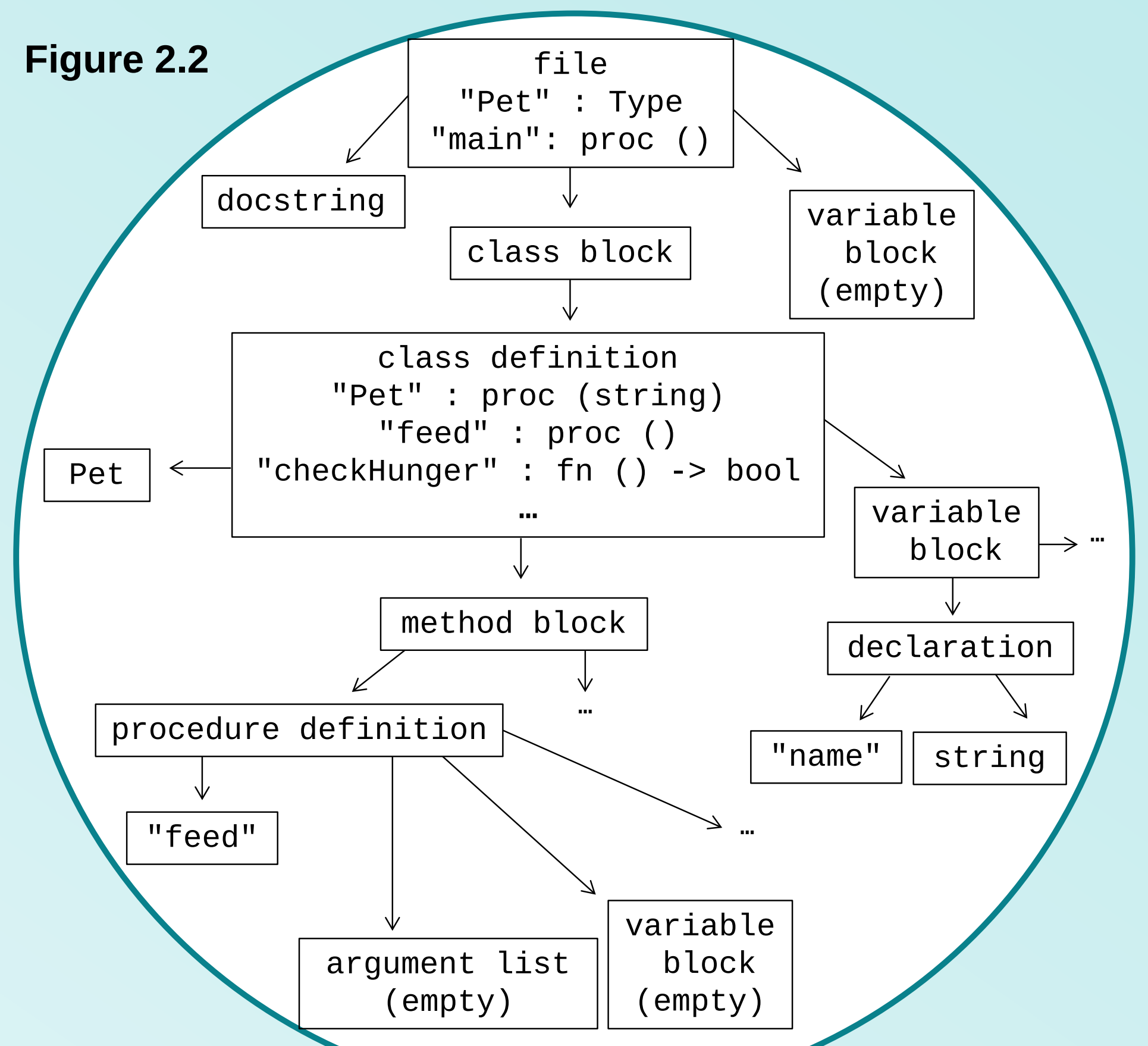

Figure 2.3

```
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

Lexical analysis is the process of splitting a sequence of characters into separate words called “tokens”. The lexer produces a token stream from CSPy source code. The token stream is passed to the parser, whose role it is to analyze the tokens in relation to the grammar (a set of rules for describing a language). From the token stream, the parser produces an abstract syntax tree (AST) containing all of the information about the syntactic structure of the source code (Figure 2.1). It is also the parser’s job to detect any syntax errors in the source code. (See Figure 3.1 for an example syntax error.)

Figure 3.1

```
[username ~]$ cspy bad_syntax.cspy
CSPy : Syntax Error
Line 3, Column 11

:: name : string
                ^
Unexpected new line.
[username ~]$
```

Figure 3.2

```
'''bad_syntax.cspy'''

:: name : string

print("Hi " + name + "!")
```

A user attempts to run their file, "`bad_syntax.cspy`" (Figure 3.2), but the variable block is missing its closing “`::`”.

Semantic Analysis

Semantic analysis is the act of ensuring all of the statements in source code are semantically correct, i.e. their meaning is consistent with the way control structures and data types are intended to be used. This type checking process is done by traversing the abstract syntax tree produced by the parser. First all variables are added to the environment of the node that holds their scope. Then each node is examined and assigned a type when appropriate. All of the nodes in a single statement are then analyzed to determine whether or not it contains any type errors. (See Figure 2.2 for an example of a parse tree with environments and types added. See Figure 3.3 for an example type error.)

Figure 3.3

```
[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 3.4

```
'''bad_type.cspy'''

:: Z : int = "Hello!" ::

print(Z)
```

A user attempts to run their file, "`bad_type.cspy`" (Figure 3.4), but the type of Z doesn't match the given value.

Translation

After a program has been type checked, if it contains no type errors, the abstract syntax tree is translated to native Python code (Figure 2.3). Because CSPy is a dialect of Python, it is much more efficient to translate CSPy and then use a Python interpreter than it is to write an interpreter just for CSPy. Therefore, the translator traverses the AST and writes a Python translation to a new file that can be executed by an interpreter. The translator also keeps track of which Python translation line numbers correspond to which CSPy source code line numbers, for use in error reporting during runtime.

Execution

After the CSPy source code is translated to native Python, the resulting Python program is run by the Python interpreter as a separate process, receiving any runtime input and returning any runtime output into the user’s terminal (Figure 2.4). If any runtime errors occur, the Python traceback object is intercepted and rewritten into a more easily understandable format for beginners. The error message includes the line from the CSPy source code where the error is occurring, as opposed to the line from the translated Python file, using the data collected by the translator. (See Figure 3.5 for an example of a standard Python error message and Figure 3.6 for a standard CSPy error message.)

Figure 3.5

```
[username ~]$ python run_error.py
Traceback (most recent call last):
  File "run_error.py", line 6, in <module>
    divide(9)
  File "run_error.py", line 4, in divide
    y = x / 0
ZeroDivisionError: integer division or modulo
by zero
[username ~]$
```

A user attempts to run their file, "`run_error.cspy`", but the file attempts to divide by 0. This error is caught and reported during runtime. The CSPy error version (Figure 3.6) aims to prioritize the error and make the message more readable.

Figure 3.6

```
[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 ~]$
```