# Programming Languages Final Project

Fahran Kamili
Caroline Chen

# Python Closures: ObjectFactory.py

- Employee class w/ private data located inside a closure ‘f’
- Apply closure ‘f’ by assigning it to ‘run’
- Closure ‘f’ includes
  - ‘Data’: attributes obtained from list comprehension project
  - ‘Cf’ which has access to ‘data’ and acts to get and set data when ‘run’ is called

Terminal Code:

```
(exec 'import ObjectFactory; employee = ObjectFactory.Employee(); employee.run
("$firstName")("Fahran"); employee.run("$lastName")("Kamili"); employee.run("$title")
("Software Engineer"); toReturn = employee.run("firstName") + " " + employee.run
("lastName") + ", " + employee.run("title")')
```

# Python Closures: ObjectFactory.py

## Terminal Code Breakdown:

```
(exec 'import ObjectFactory;

employee = ObjectFactory.Employee();

employee.run("$firstName")("Fahran"); // setter

employee.run("$lastName")("Kamili"); // setter

employee.run("$title")("Software Engineer"); // setter

toReturn = employee.run("firstName") +

" " + employee.run("lastName") +

", " + employee.run("title")') // getter
```

# Java Stream Operators

- Created Java classes Employee and Department
- Created ListFactory.py
  - builds list of emp/dept instances
- Created ListComprehension.java
  - Accepts list of employee and list of department
  - Runs stream operators corresponding to specific SQL statements and prints

Terminal Code:

```
(exec 'import ListFactory; import ListComprehension; employees = ListFactory.build("employees.txt");
departments = ListFactory.build("departments.txt"); ListComprehension.run(employees,departments)')
```

# Java Stream Operators

## Terminal Code Breakdown:

```
(exec '

import ListFactory;

import ListComprehension;

employees = ListFactory.build("employees.txt");

departments = ListFactory.build("departments.txt");

ListComprehension.run(employees,departments)')
```

# Python Lambda & List Comprehension: ListFactory.py - constructing object lists

- Imports Employee and Department Java classes (overloading constructors)
- Uses a dictionary to grab what object should be created
- Nested list comprehension to create list of instances reading from a .txt
  - Inner list = iterating through the lines of the file and processing them
  - Outer list = iterating through each processed line and passing it into class constructor

Terminal Code:

```
(ListComp (ListFactory 'employees.txt')(ListFactory 'departments.txt'))
```

# Python Lambda & List Comprehension: Mini_lisp.py

- Used lambdas in mini_lisp environment to perform a SQL statement
- SELECT dept_id, avg(salary) FROM emp GROUP BY dept_id

Terminal Code:

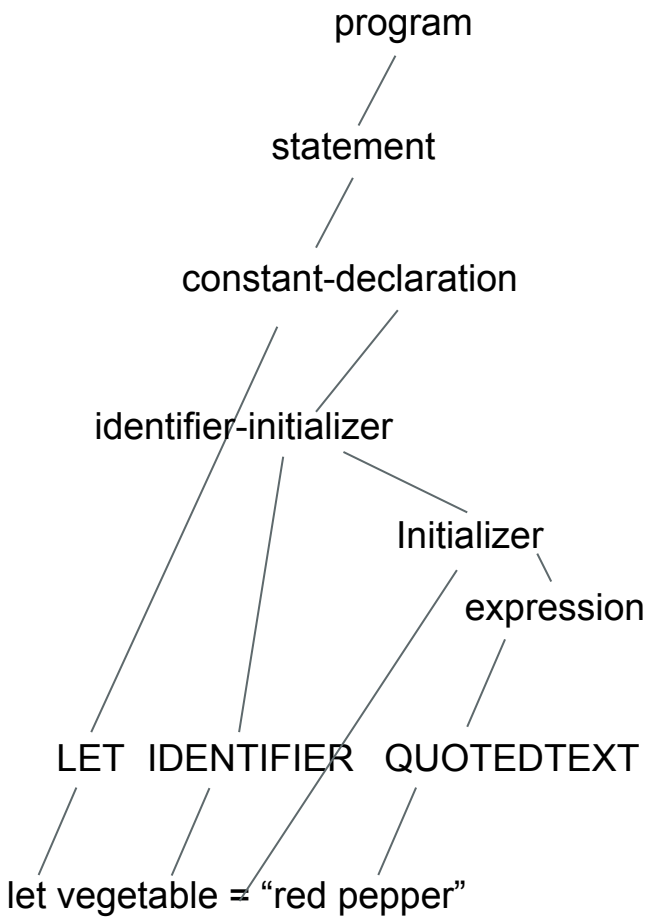
(ListComp (ListFactory 'employees.txt')(ListFactory 'departments.txt'))

# Bug Fixes in mini_lisp.py

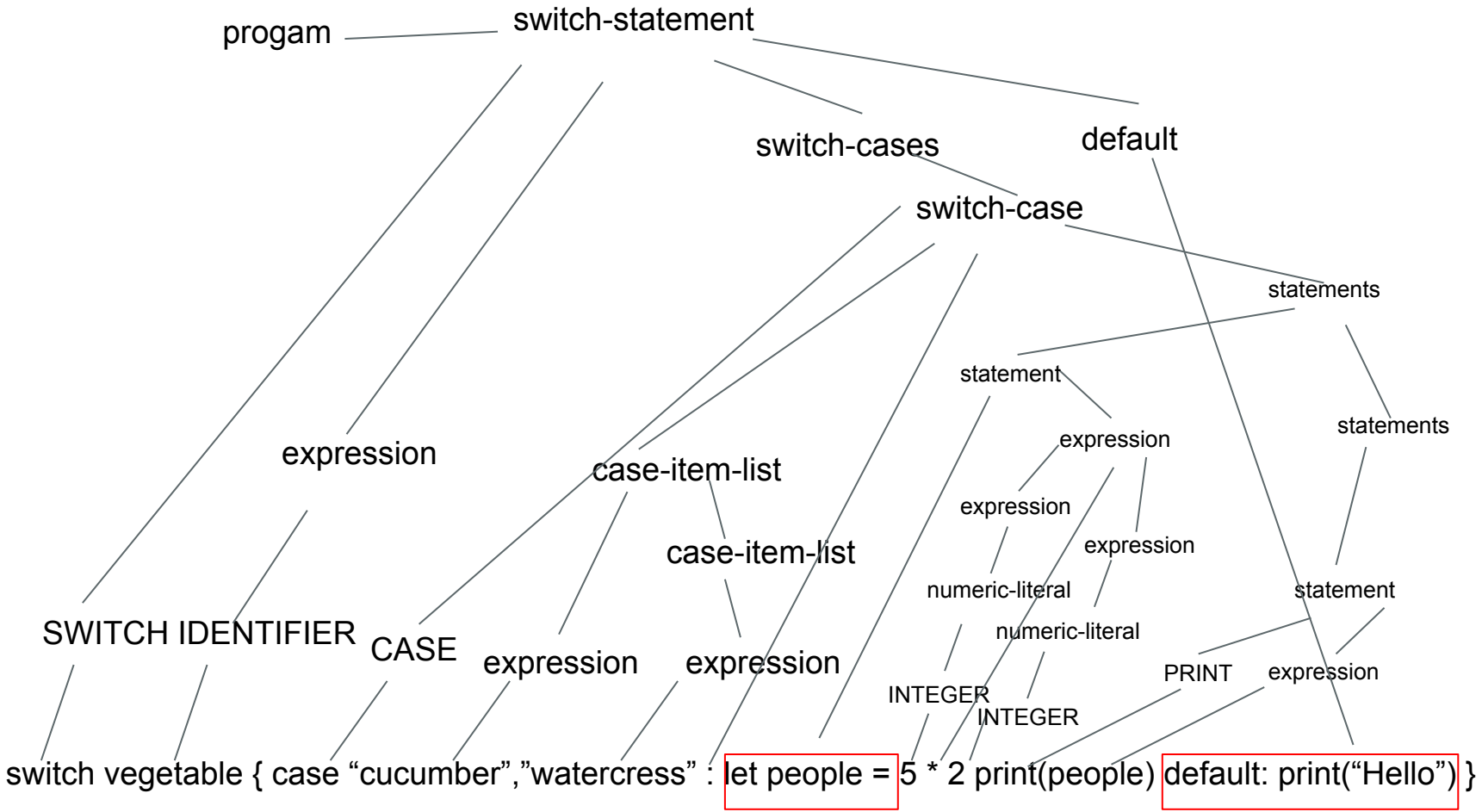
- Multiple argument support in arithmetic functions using reduce()
- Implemented cons
  - (cons pine '(maple oak))
- And/or can take any number of expressions
  - (if (and (> 3 2)(= 4 4)(< 4 1))(print 3)(print 2))
  - (or False False False True False)

# Swift parsing in PLY: Switch case

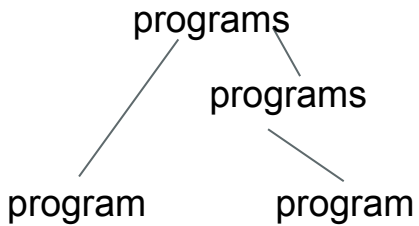
```
let vegetable = "red pepper"

switch vegetable {
        case "celery":
                print("Add some raisins and make ants on a log.")
        case "cucumber", "watercress":
                let people = 5*2
                print("That would make a good tea sandwich. For " + String(people) + " people")
        case "red pepper":
                print("Is it a spicy?")
        default:
                print("Everything tastes good in soup.")
}
```




['let', ['vegetable', '"red pepper"']]




**['switch', 'vegetable', [[['"cucumber"', '"watercress"'], [['let', ['people', ['*', 5, 2]]], ['print', 'people']]], [['default'], [['print', '"Hello"']]]]]**




**[**
['let', ['vegetable', '"cucumber"']], // **program 1**
['switch', 'vegetable', [[['"cucumber"', '"watercress"'], [['let', ['people', ['*', 5, 2]]], ['print', 'people']]], [['default'], [['print', '"Hello"']]]]]  **// program2**
**]**