Semantics (I)

Practical Expression Semantics

- Just getting the syntax (structure) of the language right doesn't mean you are generating code that works correctly. (run-time errors or unexpected results)
- You need to understand not just how to write a program, but what each part of the program actually does.

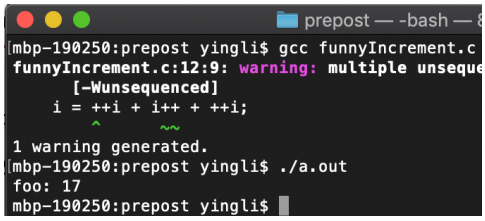
```c
//  funnyIncrement.c
//

#include <stdio.h>

void foo () {
    int i = 4;
    i = ++i + i++ + ++i;
    printf("foo: %d\n", i);
}

int main () {
    foo();
    return 0;
}
```

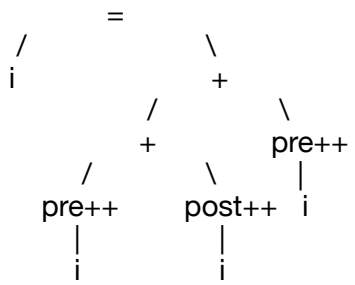
- Example:
  - Here is a C program that declares an int with value 4, and increment it and assignment it to itself. (show funnyIncrement.c)
  - What are the possible outputs of the program? [17 on my mac, 18 on d122-XX]



•The confusion here is when to execute the pre- and post-increment operators.

•If we draw a simplified concrete syntax tree, it will be a tree like this

```
yingli@davis122-917fc6:/filer/personal/yingli/cs333$ gcc funnyIncrement.c
yingli@davis122-917fc6:/filer/personal/yingli/cs333$ ./a.out
foo: 18
yingli@davis122-917fc6:/filer/personal/yingli/cs333$
```



- my mac: execute letf to right when the node is reached during tree traversal
- d122-XX: execute all pre- and post-increment before traversal

- The tree can handle operator precedence. However, it does not guarantee the order or timing of the execution of the sub-trees.
- Exciting the sub-trees in different orders can produce different results.

- C language specification does not define the behavior of the program where there is more than one pre- or post- increment operator applied to a single variable within a single expression.
- The behaviors end up being compiler specific. The semantic meaning of the program is undefined.

- How about Java?
  - Let int `x = 3; x += x++ + ++x;` This is legal in Java, so the syntax is correct.
  - What about the semantics? After the two statements have been executed, what is the value of x?.
    - The value of x is precisely defined in Java, which is 11. `x = 3 + 3 + 5.`
      - Increment and decrement operators have a higher precedence than the other mathematical operators. They are applied from left to right in the order in which they occur unless one of them has been given a higher precedence through the parentheses.

- Example:
  - `if (x++ == x++) {System.out.println("Yes");}`
    - This condition is always false in Java.
  - `if (++x == x++) {System.out.println("Yes");}`
    - This condition is always true in Java.

- Lazy evaluation
  - What is the output of the following C code? Will the function foo be invoked?
    - The foo function will not be invoked, and the output is "Expecting the second …"

```c
// LazyEval.c

#include <stdio.h>

int foo () {
     printf("Hi there!\n");
     return 1;
}
```

```c
int main () {
     if (0 && foo()) {
          printf("What's up!\n");
     }

     if (1 || foo()) {
          printf("Expecting the Thanksgiving break!\n");
     }

     return 0;
}
```

- Why the foo function wasn't invoked? [Lazy evaluation]
- Most languages now use lazy evaluation (short circuit evaluation). When evaluating boolean expressions, do it from left to right and stop as soon as you know the value of the expression. Therefore, in the example above, we don't need to call `foo()`. [C, Java, Python uses lazy evaluation]

- Example:
  - Given the following Java and Python code, which program will generate an error? [Java will generate a DivideByZero error]

```java
//LazyEval.java
public class LazyEval {
     public static void main (String[]
args) {
          int x = 0;
          int y = 4;
          if (y/x > 3 || x != 0) {
               System.out.println("Hi,
there!");
          }
     }
}
```

```python
# LazyEval.py
x = 0
y = 4

if (x != 0 and y/x > 3):
      print("Hi there!")
```

  - Both programs divide y by zero, why only the Java code generate error?
    - Because of the lazy evaluation, the second half of the expression in the python code didn't execute.

- Therefore, the order of expressions in a lazy evaluation matters.
- Why do most PLs support Lazy evaluation? [make the program more efficient]