

## Semantics

### Overview

- 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.
- Example:
  - Here is a C program that declares an int with value 4, and increment it and assignment it to itself. (show funnyIncrement.c)

```
// funnyIncrement.c
//
// Created by Ying Li on 10/6/19.
//

#include <stdio.h>

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

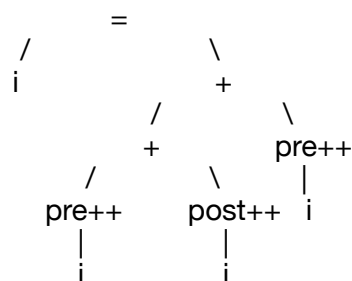
int main () {
    foo();
    return 0;
}
```

- What are the possible outputs of the program? [17 on Ying's mac, 18 on gloin, 18 on Stephanie's mac]

```
mbp-190250:prepost yingli$ gcc funnyIncrement.c
funnyIncrement.c:12:9: warning: multiple unsequenced
[-Wunsequenced]
    i = ++i + i++ + ++i;
        ^
1 warning generated.
mbp-190250:prepost yingli$ ./a.out
foo: 17
mbp-190250:prepost yingli$
```

```
gloin:~> gcc funnyIncrement.c
gloin:~> ./a.out
foo: 18
gloin:~>
```

- The confusion here is when to execute the pre-increment operators.
- If we draw a simplified concrete syntax tree, it will be a tree like this



- Ying's mac: execute left to right when the node is reached during tree traversal
- gloin: 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.
  - A left-to-right pass to generate the machine code can produce different results than a right-to left pass.
  - 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.
- Example:
    - Let `int x = 3; x += x++ + ++x;` This is legal in C and Java, so the syntax is correct.
    - What about the semantics? After the two statements have been executed, what is the value of x? And does it matter if we consider it part of a C program or part of a Java program?
      - Yes, it matters.
      - If it is a C program, the value of x depends on the compiler.
      - If it is a **Java** program, the value of x is **precisely defined**, 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.
- Example: **Lazy evaluation**
    - Give `if (false && foo()) . . .`, Is `foo()` called?
    - The answer depends on the language. Most languages now use **lazy evaluation** (short circuit evaluation). When evaluating `&&` 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]
    - Similarly, `if (true || foo())` does not require a call to `foo()`. This is important if `foo()` has side effects.
    - **Why is this important?**
      - Consider the example `if (x != 0 && y/x > 3) . . .`
        - If x is equal to 0, the lazy evaluation prevents the execution of the second expression and so avoid the DivideByZero error.
    - Therefore, **the order of expressions in a lazy evaluation matters when the expressions have side effects**.
    - Why care? [make the program more efficient]

```
//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!");
        }
    }
}
```

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

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

What are semantics? How do we define semantics?

- The **semantics** of a programming language is a **precise definition of the meaning** of any program that is **syntactically and type-wise correct**
- There are at least three ways to define the semantics
  - Operational semantics:** the **meaning** of a program **is the output** produced by a given architecture/compiler pair (pre- and post- increment in C)
  - Axiomatize semantics:** the **meaning** of a program **can be** rigorous **proven** by using a systematic logical argument (formal specification)
  - Denotational semantics:** the **meaning** of a program can be described as a **collection of meaning functions operating on the program state** (this course)

Trace Table

```
1 int main () {
2     int a, b;
3
4     a = 1;
5     b = 2;
6
7     b = a + b;
8
9     return 0;
10 }
```

| Snapshot | Before Statement | State                    |
|----------|------------------|--------------------------|
| 1        | 4                | {<a, undef>, <b, undef>} |
| 2        | 5                | {<a, 1>, <b, undef>}     |
| 3        | 7                | {<a, 1>, <b, 2>}         |
| 4        | 9                | {<a, 1>, <b, 3>}         |

## What is the program state?

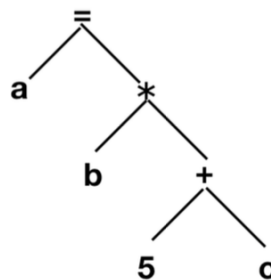
- The **state** of a program is the **binding of all variables to their current values**.
- A variable and its value can be modeled as an **ordered pair**, e.g.  $\langle i, 5 \rangle$
- A state is a **set of ordered pairs** (dictionary)

$$state = \{ \langle var1, val1 \rangle, \langle var2, val2 \rangle, \dots, \langle var_m, val_m \rangle \}$$

- A program can have more than one states.
  - A state is a snapshot of the memory of the variables used by a program. After each statement execution, the program will have a new snapshot.
- Example
  - The program calculates factorial. Show the states of this program.

## Meaning Function of Clite

- We will start with the abstract syntax tree. We will figure out the precise meaning of each node of the tree.



- Consider an assignment statement. What do assignment statements do?
  - Modify the “state” of the program or the contents of the computer’s memory.
  - How do we indicate that mathematically?
 
$$old\ state \rightarrow new\ state$$
  - This sounds like a function with input and output. An assignment statement is just a function that takes a state as input and outputs a state.
- An if statement changes the state as well. All Clite statement are just functions from  $state1 \rightarrow state2$
- We will combine all these functions into one function that takes two parameters.
 
$$M : Statement \times State \rightarrow State$$
- So
 
$$\begin{aligned} M(\text{assignment}, \text{oldstate}) &= \text{newstate} \\ M(\text{ifstatement}, \text{oldstate}) &= \text{newstate} \\ M(\text{blockstatement}, \text{oldstate}) &= \text{newstate} \end{aligned}$$
- In each case, the new state depends both on the old state and the statement being executed.
- BTW, M stands for “meaning”, as in “the meaning of this assignment statement in this state is this new state”.

- In addition to this function, we need two more:

$$M : Program \rightarrow State$$

$$M : Expression \times State \rightarrow Value$$

- A value could be any kind of r-value such as an integer, float, or any other mathematically precise object.
- Let's start implementing M for Clite.
- Note that, since we are thinking of expressions and statements in terms of the AST, and since the meanings of expressions and statements depend on the meaning of their subtrees, guess what we will have when we implement these functions? [Lots of recursion]

## Program Meaning Function

- The meaning function of Program can be implemented into this.

$$M : Program \rightarrow State$$

$$M(Program\ p) = M(p.body, InitialState(p.decpart))$$

- If we have a Program p, the implemented function should take two parameters, the body of p and the initial state of p's decpart.
- In this implementation, the InitialState is an auxiliary function that [creates a state containing all the \[variables\]\(#\) declared in p.decpart paired with their \[default values\]\(#\)](#).
- For example, if `int x = 0, y = 0;` is p.decpart in C, then `initState(p.decpart) = {<x,0>, <y,0>}`.
- Note:
  - The “**meaning**” of a Program is the **final state of the computer’s memory when the program halts**.
  - The p.body part uses recursion. Depends on the statement type in p.body, it will be implemented differently. For example, expression will have a different semantics than assignment.

```
/**
 * initialState.c
 * Ying Li
 * 10/06/2019
 */

#include <stdio.h>

int main () {
    int sum;

    for (int i = 0; i < 10; i++) {
        sum += 5;
    }

    printf("sum = %d \n", sum);

    return 0;
}
```

```

/**
 * InitialState.java
 * Ying Li
 * 10/06/2019
 */

public class InitialState {

    public static void main (String[] args) {
        int sum;

        for (int i = 0; i < 10; i++) {
            sum += 5;
        }

        System.out.printf("sum = %d \n", sum);
    }
}

```

```

/**
 * InitialState2.java
 * Ying Li
 * 10/06/2019
 */

public class InitialState2 {

    private int sum;

    public int calSum () {
        for (int i = 0; i < 10; i++) {
            sum += 5;
        }

        return sum;
    }

    public static void main (String[] args) {
        InitialState2 s = new InitialState2();
        System.out.printf("sum = %d \n", s.calSum());
    }
}

```

## InitialState in Programming Languages

- The InitialState function must be clearly defined for a language in order for programs to have meaning. [show initialState.c, initialState.java, initialState2.java]
- C does not initialize the memory space.** If not explicit initialization, the initial value of a variable is the random value stored in that memory space. (So, initialState.c gives different results every time you run it)
- Java requires the variables being initialized before using.** So initialState.java gives an error for no initialization. But if the variables are fields in a class, even without a constructor, initialState2.java won't complain, since a default constructor will be called. Default constructor will initialize the variables to the default values of their types, e.g. int is 0, float is 0.0 and string is "".

$$M(\text{Expression } e, \text{State } state) = \begin{cases} e, & \text{if } e \text{ is a Value} \\ state(e), & \text{if } e \text{ is a Varial} \\ ApplyBinary(e.op, M(e.term1, state), M(e.term2, state)), & \text{if } e \text{ is a Binari} \\ ApplyUnary(e.op, M(e.term, state)), & \text{if } e \text{ is a Unary} \end{cases}$$

## Expression Meaning Function

- We know that recursion is heavily used in the denotational semantics.
- For example, the rhs of an assignment can be a value, variable, binary or unary expression.
- So, to figure out the meaning of an assignment, we need to figure out the meaning of the rhs of the assignment.

- In Clite, an expression can be a variable, value, binary, or unary expression. *We will discuss the meaning of expression when it is a variable, value, and binary in this course.*

$$\text{Expression} = \text{Variable} \mid \text{Value} \mid \text{Binary} \mid \text{Unary}$$

- For expressions, the result is not a new state, it is just a value from a mathematical set.
- We need to break it down by the type of expression and figure out the value.
  - If the Expression is a Value, then its *meaning is* the meaning of *the Value itself*
  - If the Expression is a Variable, then its *meaning is* the *Value of the Variable in the current state*
  - If the Expression is a Binary, then the *meaning of* each of its operands *term1 and term2 is first determined*. Then *Meaning Rule of Binary Expression* determines the meaning of the expression by *applying the Operator<sub>op</sub> to the Value of those two operands*.

```
# Value
class Value:
    def __init__(self, value):
        self.value = value;

    def __str__(self):
        return str(self.value)

# Variable has the attribute name
class Variable:
    def __init__(self, name):
        self.name = name

# BinaryExpression has the attribute left,
# right, and operator
class BinaryExpression:
    def __init__(self, left, operator, right):
        self.left = left
        self.operator = operator
        self.right = right

# Maintain a dictionary that maps
# variables to their corresponding values
class State:
    def __init__(self):
        self.state = {}

    def setValue(self, var, value):
        self.state[var.name] = value

    def getValue(self, var):
        return self.state[var.name]
```

```

# expr should be a BinaryExpression
# state should be a State
# returns a value
def M_BinaryExpression(expr, state):
    if expr.operator == '+':
        val = M_Expression(expr.left, state).value + M_Expression(expr.right,
state).value
    elif expr.operator == '-':
        val = M_Expression(expr.left, state).value - M_Expression(expr.right,
state).value
    elif expr.operator == '*':
        val = M_Expression(expr.left, state).value * M_Expression(expr.right,
state).value
    elif expr.operator == '/':
        val = M_Expression(expr.left, state).value / M_Expression(expr.right,
state).value
    return Value(val)

# Meaning function of Expression
# M(Expression expr, State state)
def M_Expression (expr, state):
    if isinstance(expr, Value):
        return expr
    elif isinstance(expr, Variable):
        return Value(state.getValue(expr))
    elif isinstance (expr, BinaryExpression):
        return M_BinaryExpression(expr, state)
    else:
        print("ERROR: wrong type of expression")

def main1():
    # b + 3 * c {<b, 1>, <c, 3>}
    varb = Variable('b')
    varc = Variable('c')
    val3 = Value(3)
    expr = BinaryExpression(varb, '+', BinaryExpression(val3, '*', varc))
    state = State()
    state.setValue(varb, 1)
    state.setValue(varc, 3)
    val = M_Expression(expr, state)
    print(val)

if __name__ == "__main__":
    main1()

```

---

## Meaning Rule of Binary Expression

- The meaning rule defined a way to decide the value of an expression
- If either operand **term1** or **term2** is **undefined**, the expression is *semantically meaningless*
- If the operator is an integer operator, then integer arithmetic **add (int+)**, **subtract (int-)**, or **multiply (int\*)** perform on the integer operands, resulting in an *integer result*. If the operator

is *divide (int/)*, then the *result is the same as a mathematical divide with truncation toward zero*

- If the operator is a floating point operator, then the operands are compared with a *result of either true or false*
- meaning of an expression for a dynamic typed language

$b + 3 * c$  a floating point arithmetic using the float  $\{\langle b, 1 \rangle, \langle c, 3 \rangle\}$

operator, then the IEEE standard is operands, resulting

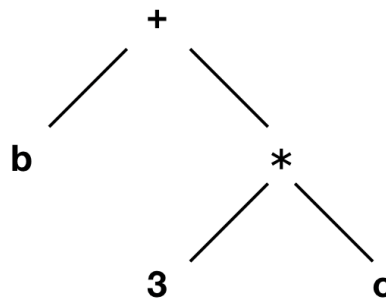
---

## Simple Interpreter

- Using the expression semantics we discussed above, we can implement a semantic interpreter to generate/check the meaning of an expression.
- Now let's implement a simple interpreter for a language with dynamic typing, no side effects, and only one scope.

---

Explain the expression in the main1 function of simple interpreter



- The AST is generated by the syntactic analyzer in the compilation.
- The structure of the tree defines the precedence of the operators.
- Then the semantic analyzer will check the semantic errors of it. It will interpret each node of the tree and evaluate it based on the semantic rules.

- Each node in the tree is a class we defined in the simple interpreter. So, we create a Value object for 3, a Variable object for each of the two variables b and c.
- Then, following the precedence and the semantic rule for binary expressions, we create a binary expression object for the right subtree  $3 * c$  and using this object coupled with the Variable object b to create another binary expression object  $b + 3 * c$ .
- In this way, the meaning of the subtree  $3 * c$  will be determined before determining the meaning for  $b + 3 * c$ .
- This also explains how the  

$$M : Expression \times State \rightarrow Value \times State$$
works here. The  
 recursion function of binary  
 meaning expression evaluate  
 the expression  $b + 3 * c$ . It will need to figure out the meaning of the left term b and the meaning of the right term  $3 * c$  first. The right term  $3 * c$  is another binary expression, whose meaning depends on the left term 3 and the right term c.
- So the process of the meaning function of binary expression to determine the meaning of the expression  $b + 3 * c$  is to depth-first traverse to the evaluate the meaning of leaves and subtrees before determining the meaning of the expression.

## Side Effects

- *Note: Our discussion above assumes there are no side effects caused by evaluating the expressions.*
- A **side effect** occurs during the evaluation of an expression if, **in addition to returning a value**, the expression **alters the state** of the program.
- A typically example is the post- and pre- increment.  $x = 1; ++x + x++;$
- *If there are side effects, the above semantics are not specific enough.*
- The way to address side effects has two steps:
  1. Update the meaning function. Instead of producing a value, it returns a value and a state.
  2. The meaning rule should be more specified. Take the binary expression for instance, the meaning rule also define the order to determine the value of two terms.
    - If the Expression is a Binary, **the meaning of term1 in the current state is first determined**, giving a value  $v1$  and a state  $s1$ . **Then the meaning of term2 in state  $s1$  is determined**, giving a value  $v2$  and a state  $s2$ . Then **Meaning Rule of Binary Expression** determines the meaning of the expression by *applying the Operator  $op$  to the resulting values  $v1$  and  $v2$  in state  $s2$* , and the resulting state is  $s2$ .
- *Note: We need to know the definition of side effect and the ways C and Java use to handle it. The implementation of side effects in meaning functions is out of the scope of the course.*

## Assignment Semantics

- We know that the rhs statement is an expression. To determine the meaning of an assignment statement, we need to know the meaning of the expression.  

$$X = \{\langle a, 1 \rangle, \langle b, 5 \rangle, \langle c, -1 \rangle\}$$

$$Y = \{\langle b, 6 \rangle, \langle d, 0 \rangle\}$$
- Now, we know the meaning of  $X \cup Y = \{\langle a, 1 \rangle, \langle b, 6 \rangle, \langle c, -1 \rangle, \langle d, 0 \rangle\}$ . Then, let's figure out the way to determine the meaning of an assignment.

- We know that the meaning function of an expression generate a new value and the meaning function of an assignment generates a new state. (Remember: Assignment is a type of statement.)

$$M : Expression \times State \rightarrow Value$$

$$M : Statement \times State \rightarrow State$$

- This means the value generated by the rhs expression is used to update the value of the lhs variable of the assignment. In the new state, the lhs variable will be associated with the new value generated by the rhs expression. Expressed this mathematically,

$$M(Assignment a, State state) = state \cup \{a.target, M(a.source, state)\}$$

- Here,  $\cup$  means overriding union.
- If we have a set  $X$  which contains three pairs,  $\{\langle a, 1 \rangle, \langle b, 5 \rangle, \langle c, 1 \rangle\}$ , and a set  $Y$  which contains two pairs,  $\{\langle b, 6 \rangle, \langle d, 0 \rangle\}$ , the overriding union of  $X$  and  $Y$  is:

The **overriding union** of  $X$  and  $Y$ , written  $X \cup Y$ , is the result of replacing in  $X$  all pairs  $\langle x, v \rangle$  whose first member matches a pair  $\langle x, w \rangle$  from  $Y$  by  $\langle x, w \rangle$  and then adding to  $X$  any remaining pairs in  $Y$

- Example
  - If  $X = \{\langle a, 1 \rangle\}$  and  $Y = \{\langle b, 2 \rangle\}$ , what is the result of overriding union of  $X$  and  $Y$ ?  
 ▶  $\{\langle a, 1 \rangle, \langle b, 2 \rangle\}$
  - If  $X = \{\}$  and  $Y = \{\langle c, 3 \rangle\}$ ?  
 ▶  $\{\langle c, 3 \rangle\}$
  - If  $X = \{\langle d, 4 \rangle\}$  and  $Y = \{\langle d, 4 \rangle\}$ ?  
 ▶  $\{\langle d, 4 \rangle\}$

```

# Assignment has the attribute target and source
class Assignment:
    def __init__(self, target, source):
        self.target = target
        self.source = source

# Maintain a dictionary that maps
# variables to their corresponding values
class State:
    def __init__(self):
        self.state = {}

    def setValue(self, var, value):
        self.state[var.name] = value

    def getValue(self, var):
        return self.state[var.name]

    def __str__(self):
        statestr = '{'
        for var in self.state:
            statestr += '<' + var + ", " + str(self.state[var]) + '>, '
        return statestr[:-2] + '}'

# Meaning function of Assignment
# M(Statement statement, State state)
def M_Assignment(statement, state):
    state.setValue(statement.target, M_Expression(statement.source,
state).value)
    return state

def main2():
    # a = b + 3 x c {<a, 5>, <b, 1>, <c, 3>}
    vara = Variable('a')
    varb = Variable('b')
    varc = Variable('c')
    val3 = Value(3)
    expr = BinaryExpression(varb, '+', BinaryExpression(val3, '*', varc))
    assignment = Assignment(vara, expr)
    state = State()
    state.setValue(vara, 5)
    state.setValue(varb, 1)
    state.setValue(varc, 3)
    newState = M_Assignment(assignment, state)
    print(newState)

if __name__ == "__main__":
    #main1()
    main2()

```

- We are going to expand the simple interpreter we built for the expression and let it be able to determine the meaning of assignment.
- Add: Assignment class, Meaning function for assignment, and let the state can be printed out pretty. Also, a main function to test them.

---

## Fun Facts of Assignment Semantics in C and Java

- C treats assignments as expression, while Java doesn't

```
/**
 * Exp.c
 * Ying Li
 * 10/15/2019
 */
#include <stdio.h>

int main () {
    int a = 1;
    int b = 0;

    if ((a = b)) {
        printf("%d == %d \n", a, b);
    } else {
        printf("%d != %d \n", a, b);
    }

    return 0;
}
```

```
/**
 * Exp.java
 * Ying Li
 * 10/15/2019
 */
public class Exp {

    public static void main (String[] args) {
        int a = 1;
        int b = 2;

        if (a = b) { //(a == b)
            System.out.printf("%d == %d \n", a, b);
        } else {
            System.out.printf("%d != %d \n", a, b);
        }
    }
}
```

- What are the output of Exp2.c? [a = 2, b = 2, c = 2; a = 0, b = 0, c = 3 ]

- In assignments  $a = b = c$  evaluates  $c$ , assigns that to  $b$ , then assigns that to  $a$ .
- In expressions  $c = 3 + (a = b)$  assigns the value of  $b$  to  $a$  and then adds 3 to the value and assigns it to  $c$ .

```

/**
 * Exp2.c
 * Ying Li
 * 10/15/2019
 */

#include <stdio.h>

int main () {
    int a = 1;
    int b = 0;
    int c = 2;

    a = b = c;
    printf("a = %d, b = %d, c = %d \n", a, b, c);

    a = 1;
    b = 0;
    c = 2;

    c = 3 + (a = b);
    printf("a = %d, b = %d, c = %d \n", a, b, c);

    return 0;
}

```

- When assigning a value, C and Java use different semantics.
  - C uses **copy semantics**, which means the assignment **takes the value generated by the right side and puts a copy of it into the memory location bound to the variable on the left side**.
  - Java uses **copy semantics on primitive types**, but for **object type**, it uses **reference semantics**.
  - **Reference semantics** means that the **data type Object** is not **bound** to the memory location where the Object's data is located, but **to a memory location that holds a reference to the object**. Therefore, the assignment copies only the reference to the Object. [Show Ref.java]

```
/**
 * http://docs.oracle.com/javase/6/docs/api/java/
 * lang/Object.html#toString%28%29
 * http://rosettacode.org/wiki/
 * Address_of_a_variable#Java
 * getClass().getName() + '@' +
 * Integer.toHexString(hashCode())
 * hashCode() is typically implemented by
 * converting the internal address of the
 * object into an integer. So it reflects at least
 * part of the address of an object
 * System.identityHashCode() accesses the original
 * hash code
 */
```

```
public class Ref {

    public static void main(String[] args) {
        Integer i = new Integer(5);
        Integer a = i;
        Integer b = i;
```

$$M(\text{Conditional } c, \text{State } state) = \begin{cases} M(c.\text{thenbranch}, state) & \text{if } c \text{ is true} \\ M(c.\text{elsebranch}, state) & \text{if } c \text{ is false} \end{cases}$$

```
        System.out.println("i: " + i + " " +
        Ref.objToString(i));
        System.out.println("a: " + a + " " +
        Ref.objToString(a));
        System.out.println("b: " + b + " " +
        Ref.objToString(b));
    }
}
```

```
// mimic Object toString
public static String objToString (Object o) {
    String s = null;

    if (o != null) {
        s = o.getClass().getName() + "@" +
        Integer.toHexString(System.identityHashCode(o));
    }
}
```

```
/**
 * Exp3.c
 * Ying Li
 * 10/15/2019
 */
```

```
#include <stdio.h>
```

```
typedef struct {
    char name[20];
    int age;
} Dog;
```

```
int main () {
    int a = 1;
    int b = 0;
```

```
    a = b;
    printf("a (%p): %d \n", &a, a);
    printf("b (%p): %d \n", &b, b);
```

```
    Dog dog1 = {"Toto", 2};
    Dog dog2 = dog1;
    printf("dog2 %s, %d\n", dog2.name,
    dog2.age);
    printf("dog1 @ %p, dog2 @ %p \n", &dog1,
    &dog2);
```

```
    return 0;
}
```

## Conditional Semantics

- In addition to the assignment statement, conditional statement (if statement) is another type of statement.
- Its meaning function also takes in a state and generate another state.

$$M : \text{Statement} \times \text{State} \rightarrow \text{State}$$

• A conditional statement has three elements: an expression, then branch statement, and else branch statement.

$$\text{Conditional} = \text{Expression test}; \text{Statement thenbranch}, \text{elsebranch}$$

- The **meaning of a conditional depends on the meaning of the expression.**
  - If the expression is true, the meaning of the conditional is the meaning of the then branch.
  - If the expression is false, the meaning of the conditional is the meaning of the else branch.

```

# expr should be a BinaryExpression
# state should be a State
# returns a value
def M_BinaryExpression(expr, state):
    if expr.operator == '+':
        val = M_Expression(expr.left, state).value + M_Expression(expr.right,
state).value
    elif expr.operator == '-':
        val = M_Expression(expr.left, state).value - M_Expression(expr.right,
state).value
    elif expr.operator == '*':
        val = M_Expression(expr.left, state).value * M_Expression(expr.right,
state).value
    elif expr.operator == '/':
        val = M_Expression(expr.left, state).value / M_Expression(expr.right,
state).value
    elif expr.operator == '>':
        val = M_Expression(expr.left, state).value > M_Expression(expr.right,
state).value
    elif expr.operator == '<':
        val = M_Expression(expr.left, state).value < M_Expression(expr.right,
state).value
    elif expr.operator == '==':
        val = M_Expression(expr.left, state).value == M_Expression(expr.right,
state).value
    elif expr.operator == '!=':
        val = M_Expression(expr.left, state).value != M_Expression(expr.right,
state).value
    return Value(val)

def main3():
    # if (a < b) {
    #   min = a;
    # }
    # else {
    #   min = b;
    # }
    # {<min, 0>, <a, 3>, <b, 5>}
    varmin = Variable('min')
    vara = Variable('a')
    varb = Variable('b')
    state = State()
    state.setValue(varmin, 0)
    state.setValue(vara, 3)
    state.setValue(varb, 5)
    expr = BinaryExpression(vara, '<', varb)
    thenbranch = Assignment(varmin, M_Expression(vara, state))
    elsebranch = Assignment(varmin, M_Expression(varb, state))
    cond = Conditional(expr, thenbranch, elsebranch)
    newState = M_Statement(cond, state)
    print(newState)

if __name__ == "__main__":
    #main1()
    #main2()
    main3()

```

- To extend our simple interpreter for the conditional statement, we need to add conditional class, the meaning function to statements including assignments, conditionals, and extendible to other statement types, the meaning function of conditional, and a main function to test them. Remember to extend the meaning function for binary expression to support relational operators: >, <, ==, and !=. (Part of HW6)

```
# Conditional has three attribute: condition,
# thenbranch, and elsebranch
class Conditional:
    def __init__(self, condition, thenbranch, elsebranch):
        self.condition = condition
        self.thenbranch = thenbranch
        self.elsebranch = elsebranch

# Meaning function of Statement
# M(Statement statement, State state)
def M_Statement (statement, state):
    if isinstance(statement, Assignment):
        return M_Assignment(statement, state)
    elif isinstance(statement, Conditional):
        return M_Conditional(statement, state)
    else:
        print("ERROR: wrong type of statement")

# Meaning function of Contional
# M(Statement statement, State state)
def M_Conditional (statement, state):
    if M_Expression(statement.condition, state).value:
        state = M_Statement(statement.thenbranch, state)
    else:
        state = M_Statement(statement.elsebranch, state)
    return state
```

- Exercise: write a main function to let the simple interpreter determine the meaning of the following conditional statement. {<min, 0>, <a, 3>, <b, 5>}

```
if (a < 5) {
    if (b > 3) {
        min = 100;
    }
    else {
        min = 50;
    }
}
```

- We need another type of statement here, **Skip**, for the then branch of the outer conditional statement if (a < 5).
- The else branch of the inner conditional statement if (b > 3) has nothing. From the perspective of a compiler/interpreter, this is a [skip statement which takes in a state and returns the same state without modifying anything](#).

```
# Skip
class Skip:
    pass

# Meaning function of Skip
# M(Statement statement, State state)
def M_Skip (statement, state):
    return state;
```

- Then, we need to update the meaning function of statement to let it be able to handle Skip.

```
# Meaning function of Statement
# M(Statement statement, State state)
def M_Statement(statement, state):
    if isinstance(statement, Assignment):
        return M_Assignment(statement, state)
    elif isinstance(statement, Conditional):
        return M_Conditional(statement, state)
    elif isinstance(statement, Skip):
        return M_Skip(statement, state)
    else:
        print("ERROR: wrong type of statement")
```

- We can then implement the main function as

$$M(\text{Block } b, \text{State } state) = \begin{cases} state, & \text{if } b = \{\} \\ M(b_n, M(b_{n-1}, \dots, M(b_1, state) \dots)), & \text{if } b = \{b_1, b_2, \dots, b_n\} \end{cases}$$

```
def main4():
    varmin = Variable('min')
    vara = Variable('a')
    varb = Variable('b')
    state = State()
    state.setValue(varmin, 0)
    state.setValue(vara, 3)
    state.setValue(varb, 5)
    inexpr = BinaryExpression(varb, '>', Value(3))
    outexpr = BinaryExpression(vara, '<', Value(5))
```

---

## Block Semantics

- The conditional and skip semantics work well for the above if statement in which each branch only contains one statement. However, it's more common in programs that a branch contains more than one statement, which we call a block.
- A **block** is a **set of statements**.

$$\text{Block} = \text{Statement}^*$$

- We need a meaning function for a block.
  - If the set is **empty**, the block has no statement. So the **meaning of the program is not changed** by the block.
  - **Otherwise**, the **meaning** of the block **is the aggregated results of all statements** in the block.
- The implementation of the meaning function
  - If the set is **empty**, the **state does not change**.
  - If there are **n statements** in the block, the meaning function of each statement is called. The **inner most meaning function is the first statement** in the block, and the **outer most is the last statement**.
  - Depending on the type of each statement, different meaning functions are called.

```
def main5():
    # if (a < 5) {
    #   min = 100;
    #   b = 0;
    # }
    # else {
    #   min = 50;
    #   b = 1;
    # }
    # {<min, 0>, <a, 3>, <b, 5>}
    varmin = Variable('min')
    vara = Variable('a')
    varb = Variable('b')
    state = State()
    state.setValue(varmin, 0)
    state.setValue(vara, 3)
    state.setValue(varb, 5)
    expr = BinaryExpression(vara, '<', Value(5))
    ifblkstatmnt1 = Assignment(varmin, Value(100))
    ifblkstatmnt2 = Assignment(varb, Value(0))
    elsblkstatmnt1 = Assignment(varmin, Value(50))
    elsblkstatmnt2 = Assignment(varb, Value(1))
    ifblk = Block([ifblkstatmnt1, ifblkstatmnt2])
    elsblk = Block([elsblkstatmnt1, elsblkstatmnt2])
    cond = Conditional(expr, ifblk, elsblk)
    newState = M_Statement(cond, state)
    print(newState)

if __name__ == "__main__":
    #main1()
    #main2()
    #main3()
    #main4()
    main5()
```

- To extend the simple interpreter, we need a block class, a meaning function for block, a main function to test the code, and extend the meaning function of statement to support blocks.

---

## Loop Semantics

- Another type of statements that heavily relies on blocks is loop.
- A typical while loop is composed of an expression, test, and a loop body, a block of statements.

$$M(\text{Loop } l, \text{State } state) = \begin{cases} M(l, M(l.\text{body}, state)), & \text{if } M(l.\text{test}, state) \text{ is true} \\ state, & \text{otherwise} \end{cases}$$

*Loop = Expression test; Statement body*

- The **meaning of a while loop depends on the meaning of the expression**.
  - If the expression is **true**, it is the **meaning of the body** of the loop.
  - Otherwise, the **meaning is the state**.

- Note that, **l.body** here **is a statement**. It can be an assignment, an expression, a conditional, a skip, or a block.

```
# Loop
class Loop:
    def __init__(self, test, body):
        self.test = test
        self.body = body

# Meaning function of Loop
# M(Statement loop, State state)
def M_Loop(loop, state):
    if M_Expression(loop.test, state).value == False:
        return state
    return M_Loop(loop, M_Statement(loop.body, state))

# Meaning function of Statement
# M(Statement statement, State state)
def M_Statement(statement, state):
    if isinstance(statement, Assignment):
        return M_Assignment(statement, state)
    elif isinstance(statement, Conditional):
        return M_Conditional(statement, state)
    elif isinstance(statement, Skip):
        return M_Skip(statement, state)
    elif isinstance(statement, Block):
        return M_Block(statement, state)
    elif isinstance(statement, Loop):
        return M_Loop(statement, state)
    else:
        print("ERROR: wrong type of statement")

def main6():
    # while (a < 5) {
    #   a = a + 1;
    #   b = b - 1;
    # }
    # {<a, 1>, <b, 10>}
    vara = Variable('a')
    varb = Variable('b')
    state = State()
    state.setValue(vara, 1)
    state.setValue(varb, 10)
    test = BinaryExpression(vara, '<', Value(5))
    expr1 = BinaryExpression(vara, '+', Value(1))
    expr2 = BinaryExpression(varb, '-', Value(1))
    assignment1 = Assignment(vara, expr1)
    assignment2 = Assignment(varb, expr2)
    block = Block([assignment1, assignment2])
    loop = Loop(test, block)
    newState = M_Loop(loop, state)
    print(newState)

if __name__ == "__main__":
    #main1()
    #main2()
    #main3()
    #main4()
    #main5()
    main6()
```

- It also means that it is okay that if there is an assignment, expression, or conditional after the expression *without using curly braces*.
- If there are more than one statement, curly braces should be used to form a block.
- To extend the simple interpreter, we need a loop class, a meaning function for loop, a main function to test the code, and extend the meaning function of statement for the loop.
- For loop semantics
  - The abstract syntax for a for loop requires a number of different fields: an initial statement, an expression, the body statement, and the post statement.

*For = Statement initial; Expression test; Statement body; Statement post*

- The meaning of the for loop is a combination of things.
  - The initial statement modifies the state prior to evaluation of the test expression.
  - The body and post statements modify the state only if the test expression is true.
  - The post statement modifies the state after the application of the body to the state.
- The for loop syntax in C/Java is special, because commas are used to differentiate different statements in the initial and post sections. However, the semantic interpretation is to treat them as blocks, in which case they are Statements.
- Because the initial section executes only once, we have to use two functions to represent the meaning of the for loop.

$$M(\text{For } f, \text{State } state) = M(f, M(f.\text{initial}, state))$$

$$M(\text{For } f, \text{State } state) = \begin{cases} state, & M(f.\text{test}, state) = \text{False} \\ M(f, M(f.\text{post}, M(f.\text{body}, state))), & \text{otherwise} \end{cases}$$

## I/O Semantics

- I/O includes file I/O and standard I/O (read from keyboard and write to screen).
  - Some languages have I/O mechanisms built into the language syntax (PHP, Prolog, and Python)
    - In Python, we can use `print`, `raw_input`, `open`, `close`, `read`, `write`, etc. directly.
  - Many languages use a built-in library of functions to handle all I/O (C, Fortran, Lisp)
    - In C, input and output are not part of the C language itself, but defined in standard library `<stdio.h>`.

## C Standard In and Standard Out

- `int getchar (void)` // read one character at a time from the standard input (keyboard, normally)
- `int putchar (int)` // put the character `int` to the standard output (screen by default)
- Change the input/output direction
  - If a program `prog` uses `getchar`, the the command line `prog < infile` causes `prog` to read characters from `infile` instead.
  - The command line `otherprog | prog` runs the two programs `otherprog` and `prog`, and pipes the standard output of `otherprog` into the standard input for `prog`.
  - If `prog` uses `putchar`, `prog > outfile` will write the standard output to `outfile` instead.
- Show `lower.c` and run it on terminal

```
/**
 * lower.c
 * Convert input to lower case
 */

#include <stdio.h>
#include <ctype.h>

int main () {
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

```
/**
 * test.c
 * A simple code to generate a upper case string
 */

#include <stdio.h>

int main () {
    printf("WORLD\n");
    return 0;
}
```

```
/**
 * test.txt
 */
H
E
L
L
O
```

```
$ gcc lower.c -o lower
$ ./lower
ABCD (press enter)

The output is
$ ./lower
ABCD
abcd (ctl+D to EOF)

Show test.txt and run
$ ./lower < test.txt
h
e
l
l
o

Show test.c and run pipes
$ gcc test.c -o test
$ ./test | ./lower
world

$ ./lower < test.txt > rlt.txt
$ cat rlt.txt
h
e
l
l
o

Run the code like following, ask the output
$ ./lower < test.txt | ./lower
h
e
l
l
o
```

```
/**
 * print.c
 */

#include <stdio.h>

int main () {
    char c = 97;
    printf("%c\n", c); //a

    int a = 15, b = 9;
    printf("%X, %o\n", a, b); //F, 11

    return 0;
}
```

---

## Formatted Output - printf

- `printf` converts, formats, and prints its arguments on the standard output under control of the format. [Show `print.c`, and ask the output.]

---

## Formatted Input

- `int scanf (char *format, arg1, arg2, ...)`
  - reads characters from the standard input
  - interprets them according to the specification in `format`
  - stores the results through the remaining arguments, each of which must be a pointer
  - stops when it exhausts its format string, or when some input fails to match the control specification
  - returns the number of successfully matched and assigned input items; can be used to decide how many items were found.
- `int sscanf (char *string, char *format, arg1, arg2, ...)`
  - scans the string according to the format in `format` and stores the resulting values through `arg1`, `arg2`, etc. These arguments must be pointers.
  - blanks or tabs are not ignored
- Show `rudimentary.c`, ask what the output of the program?

```
/**
 * rudimentary.c
 */

#include <stdio.h>

int main () {
    double sum, v;

    sum = 0;
    // compile time warning and run time segment fault; should be scanf("%lf", &v)
    while (scanf("%lf", v) == 1)
        printf("\t%.2f\n", sum +=v);
    return 0;
}
```

---

## File Access

- `fopen`:
  - Before it can be read or written, a file has to be opened

- fopen takes an external name like x.c does some housekeeping and negotiation with the operating system, and returns a pointer to be used in subsequent reads or writes of the file.
- **FILE \*fopen(char \*name, char \*mode)**
- The pointer called the [file pointer](#)
  - [Points to a structure that contains information about the file](#), such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors or end of file have occurred.
  - Users don't need to know the details, since the definitions obtained from `<stdio.h>` include a structure declaration called FILE.
  - The [only declaration needed for a file pointer](#) is exemplified by **FILE \*fp;**
  - Notice that **FILE is a type name**, like `int`, not a structure tag; it is defined with a typedef.
- The call to fopen is **fp = fopen(name, mode)**
- [mode](#)
  - a [character](#) string [indicates how](#) one intends to [use the file](#)
  - [read \("r"\)](#), [write \("w"\)](#), and [append \("a"\)](#)
  - some systems distinguish between text and binary files; for the latter, a "b" must be appended to the mode string. E.g. "rb" means reading binary file.
  - If [a file that does not exist is opened for writing or appending, it is created](#) if possible.
  - Opening an existing file for [writing causes the old contents to be discarded](#), while opening for [appending preserves them](#).
  - Trying to [read a file that does not exist is an error](#). Trying to read a file when you don't have permission is also an error. If there is any error, open will return NULL.
- `int fclose(FILE *fp)`
  - [Free the file pointer for another file](#)
  - Since most operating systems have some limit on the number of files that a program may have open simultaneously, it is a good idea to free the file pointers when they are no longer needed.
- For [formatted input or output of files](#), the functions `fscanf` and `fprintf` may be used.
  - Identical to `scanf` and `printf`, except that the first argument is a file pointer
 

```
int fscanf(FILE *fp, char *format, ...)
```

```
int fprintf(FILE *fp, char *format, ...)
```
  - Show `formatIO.c` and `input.txt`, run `./formatIO input.txt output.txt`, and show the `output.txt`.
  - Show `formatIO2.c`, run `./formatIO2` with the same `input.txt`, and show the `output.txt`. Ask why the outputs are different. [%d vs %c]
  - Read and write as a stream
 

```
size_t fread(void *array, size_t size, size_t count, FILE *fp);
```

```
size_t fwrite(void *array, size_t size, size_t count, FILE *fp);
```

    - `array`: a block of memory with the minimum size of `size*count` bytes
    - `size`: the size in bytes of each element to be read
    - `count`: number of elements
    - `return`: the total number of elements successfully read

|                                                                                                                                                                                                                                                                                                                                        |                                                                                       |                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|----------------------------------|
| <pre> /**  * formatted  * file I/O  *  * formatIO2.c  */  #include &lt;stdio.h&gt;  int main (int argc, char* argv[]) {     FILE *fp1, *fp2;     unsigned char value[20];      fp1 = fopen(argv[1], "r");     fp2 = fopen(argv[2], "w");     while (fscanf(fp1, "%c", value) == 1) {         fprintf(fp2, "%s", value);     } } </pre> | <pre> \$ ./formatIO input.txt output.txt \$ cat output.txt 1 10 100 1000 10000 </pre> | <pre> 1 10 100 1000 10000 </pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|----------------------------------|

- ▶ Show streamIO.c and input.txt, run ./streamIO input.txt output.txt, and show the output.
- ▶ Show streamIO2.c and input2.txt, run ./streamIO2 input2.txt output.txt, and show the output. Ask why the outputs are different.[value[20] is not big enough, as streamIO reads all characters include invisible ones '\n'].

- Unformatted file I/O
  - int `getc(FILE *fp)`
    - ▶ return the next character from a file referred to by fp
    - ▶ it returns EOF for end of file or error
  - int `putc(int c, FILE *fp)`
    - ▶ writes the character c to the file fp and return the character written, or EOF if an error occurs
- Fun facts
  - When a C program is started, the operating system environment is responsible for opening three files and providing pointers for them. These files are the standard input, the standard output, and the standard error; the corresponding file pointers are called stdin, stdout, and stderr, and are declared in <stdio.h>.
  - getchar and putchar can be defined in terms of getc, putc, stdin, and stdout as follow:

```

/**
 * streamI0.c
 */

#include <stdio.h>

int main (int argc, char *argv[]) {
    FILE *fp1, *fp2;
    int value[5];

    fp1 = fopen(argv[1], "r");
    fp2 = fopen(argv[2], "w");

    fread(value, sizeof(int), 5, fp1);
    fwrite(value, sizeof(int), 5, fp2);

    fclose(fp1);
    fclose(fp2);

    return 0;
}

```

input.txt

```

1
10
100
1000
10000

```

```

$ ./streamI0 input.txt output.txt
$ cat output.txt
1
10
100
1000
10000

```

```

/**
 * streamI02.c
 * Look into the array memory
 */

#include <stdio.h>

int main (int argc, char *argv[]) {
    FILE *fp1, *fp2;
    unsigned char value[20];

    fp1 = fopen(argv[1], "r");
    fp2 = fopen(argv[2], "w");

    fread(value, sizeof(int), 5, fp1);
    /*for (int i = 0; i < 20; i++) {
        printf("%d\n", *(value+i));
    }*/
    fwrite(value, sizeof(int), 5, fp2);

    fclose(fp1);
    fclose(fp2);

    return 0;
}

```

input2.txt

```

1000
2000
3000
4000
5000

```

```

$ ./streamI02 input2.txt output.txt
$ cat output.txt
1000
2000
3000
4000

```

- ```
#define getchar() getc(stdin)
#define putchar(c) put((c), stdout)
```
- The file pointers `stdin` and `stdout` are objects of type `FILE *`. They are **constants**, however, not variables, so it is **not possible to assign to them**.
  - Example:
    - Implement a C program to realize the command “cat”
    - “cat” can be used in two ways:
      - `$ cat file1 file2`: print the contents of file1 and file2 to the terminal
      - `$ cat`: print the standard input contents to the standard output

```
[mbp-190250:code yingli$ cat test1.txt
A
B
C
D
[mbp-190250:code yingli$ cat test1.txt test2.txt
A
B
C
D
1
2
3
4
[mbp-190250:code yingli$ cat
HELLO
HELLO
WORLD
WORLD
mbp-190250:code yingli$
```

- To implement it, we can either use a conditional statement to choose between `getChar` and `getC` to read the contents from files or standard input or use the file pointers (use regular file pointers for input files and use `stdin` to read from standard input)
- The following program gives an example to use file pointers coupled with `getC` and `putC` to implement “cat”

---

## Exception Semantics

- Error Handling
  - Function **return value**
    - C has the **convention** that a function that **executes properly should return a 0** if it is not using the function’s return value for something else.
  - **Exceptions are thrown/caught**. This disrupts the normal flow of execution.
  - **Assertions**. Check the correctness of assumptions during run-time
  - **Signals**. A software interrupt delivered to a process. OS uses it to report exceptional situations to an executing program (e.g., references to invalid memory addresses).
- Exceptions
  - The formal semantics of exceptions is out of the scope of this course.

```

/**
 * cat: concatenate file
 */

#include <stdio.h>

/*filecopy: copy file ifp to file ofp */
void filecopy (FILE *ifp, FILE *ofp) {
    int c;

    while ((c = getc(ifp)) != EOF)
        putc(c, ofp);
}

int main (int argc, char *argv[]) {
    FILE *fp;

    if (argc == 1) // no args; copy standard input
        filecopy (stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: can't open %s\n", *argv);
                return 1;
            }
            else {
                filecopy(fp, stdout);
                fclose(fp);
            }

    return 0;
}

```

- We will describe the flow of execution when an exception is encountered.
  - C++ example: The exceptions can be any type (std::exception, int, etc.). The throw statement essentially calls the catch block. [Show [trycatch.cc](#)]
- 
- A throw within a function can be caught in the parent function. [Show [trycatchPC.cpp](#)]

```

#include <stdio.h> //stdio.h in C

int main (int argc, char *argv[]) {
    for (;;) {
        int q;

        try {
            printf("Enter a number (0..9): ");
            int k = scanf("%d", &q); // read formatted from stdin

            if (k == 0) {
                scanf("%*s"); //read the value but ignore
                throw "value is not a number";
            }

            //this part is unnecessary, just show different types of exceptions
            if (q < 0) {
                throw 3.0;
            }

            if (q < 0 || q > 9) {
                throw q;
            }
        }
        catch (const char *s) {
            printf("Error: %s\n", s);
            continue;
        }
        catch (int v) {
            printf("Error: number %d is out of range (0..9)\n", v);
            continue;
        }
        catch (...) {
            printf("Unsepcified error\n");
            continue;
        }

        if (q == 0)
            break;
    }

    return 0;
}

```

```

$ ./a.out
Enter a number (0..9): a
Error: value is not a number
Enter a number (0..9): 10
Error: number 10 is out of range (0..9)
Enter a number (0..9): -1
Unsepcified error
Enter a number (0..9): 0
$

```

- Java example: **A function can throw an exception it does not catch.** To do this, it **must declare the property of the exception** to calling functions **by including a throw clause in its preamble.**
  - Like C++, **Java permits multiple catch blocks differentiated by their argument.**
  - The try/catch structure can also include a **block labeled with finally**, which has **no arguments. Code in the finally section is always executed** whether or not the code throws an exception. It is executed even if one of the try/catch blocks calls break or return. [Show trycatch.java, ask the outputs]
- Python example: If an exception occurs in the try statement, execution moves to the except block if the exception matches one of the exceptions listed.

```

#include <stdio>

int childFunc () {
    int q;

    try {
        printf("Enter a number (0..9): ");
        int k = scanf("%d", &q);

        if (k == 0) {
            scanf("%*s");
            throw "value is not a number";
        }
        if (q < 0 || q > 9) {
            throw q;
        }
    }
    catch (int v) {
        printf("Error: number %d is out of range (0..9)\n", v);
        return v;
    }

    return q;
}

int main () {
    int val;

    for (;;) {
        try {
            val = childFunc();
        }
        catch (const char *s) {
            printf("Error: %s\n", s);
            continue;
        }

        if (val == 0)
            break;
    }

    return 0;
}

```

```

$ g++ trycatchPC.cc
$ ./a.out
Enter a number (0..9): -1
Error: number -1 is out of range (0..9)
Enter a number (0..9): 10
Error: number 10 is out of range (0..9)
Enter a number (0..9): 9
Enter a number (0..9): a
Error: value is not a number
Enter a number (0..9): 0
$

```

- The syntax **permits multiple except cases**. An except block can also list

**multiple exceptions inside a tuple**, e.g. (RuntimeError, TypeError, NameError, ValueError).

- An **except** block with no arguments catches all exceptions not explicitly caught by other except cases.
- The equivalent of throw in Python is the **raise** keyword. The raise statement **takes the exception class as an argument**. [Show trycatch.py]
- Python also includes a finally clause like Java.

```
def demo1():
    a = 0

    while a == 0:

        try:
            s = input('enter a number: ')
            val = int(s)

            except ValueError:
                print("not a valid number")
                continue

            if val == 0:
                a = 1

        print("terminating")

def demo2_helper():
    raise 1

def demo2():
    try:
        demo2_helper()
    except:
        print("demo 2 catching the error")

demo1()
#demo2()
```

Comment out demo2() and run the code

```
$ python3 trycatch.py
enter a number: a
not a valid number
enter a number: 1
enter a number: 0
terminating
```

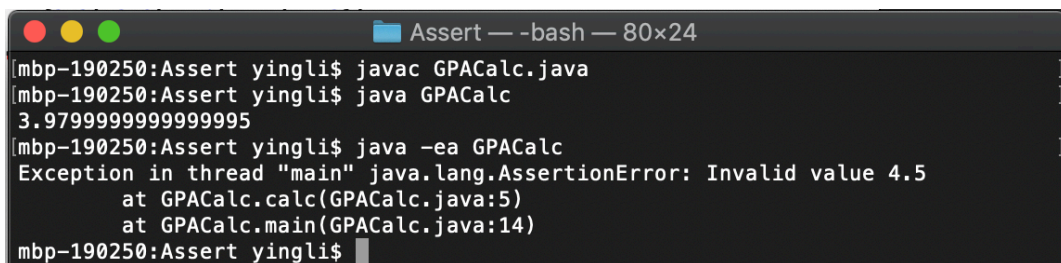
Comment out raise inside the except of demo2() function, and run again

```
$ python3 trycatch.py
demo 2 catching the error
```

- Assertions
  - Statements used to [check the correctness of assumptions](#) made in a program [during run-time](#).
  - Debug tool
  - C example:

```
void assert (int expression);
```

- If the expression meaning is 0 (false), the expression, source code filename, and line number are sent to standard error, and then the abort function is invoked.



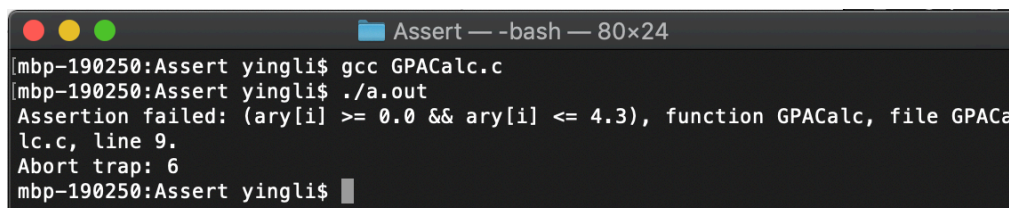
```
mbp-190250:Assert yingli$ javac GPACalc.java
mbp-190250:Assert yingli$ java GPACalc
3.9799999999999995
mbp-190250:Assert yingli$ java -ea GPACalc
Exception in thread "main" java.lang.AssertionError: Invalid value 4.5
    at GPACalc.calc(GPACalc.java:5)
    at GPACalc.main(GPACalc.java:14)
mbp-190250:Assert yingli$
```

-Java

example:

```
assert expression;
assert expression1 : expression2;
```

- expression1: boolean expression; If false, JVM throws AssertionError.
- expression2: an expression that has a value.
- `java -ea` : enable assertions

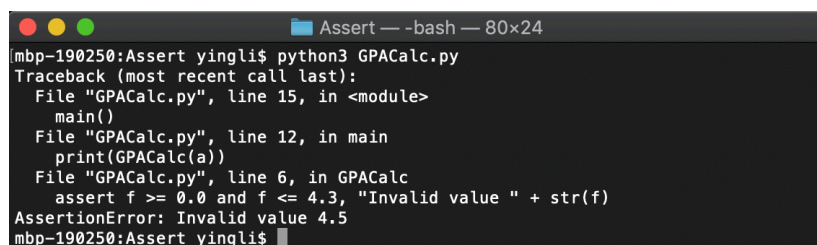


```
mbp-190250:Assert yingli$ gcc GPACalc.c
mbp-190250:Assert yingli$ ./a.out
Assertion failed: (ary[i] >= 0.0 && ary[i] <= 4.3), function GPACalc, file GPACalc.c, line 9.
Abort trap: 6
mbp-190250:Assert yingli$
```

- Python example

```
assert condition, error_message(optional)
```

- condition: boolean expression
- error\_message: printed on terminal if AssertionError



```
mbp-190250:Assert yingli$ python3 GPACalc.py
Traceback (most recent call last):
  File "GPACalc.py", line 15, in <module>
    main()
  File "GPACalc.py", line 12, in main
    print(GPACalc(a))
  File "GPACalc.py", line 6, in GPACalc
    assert f >= 0.0 and f <= 4.3, "Invalid value " + str(f)
AssertionError: Invalid value 4.5
mbp-190250:Assert yingli$
```

- Signals

- In C, signals can be used to [handle exceptions](#), and it has a [library to handle signals](#). These are [not the same as exceptions](#) (though they could be caused by similar errors, such as dereferencing a null pointer).
- An example of a signal that is not an exception is a key-board interrupt. This is an external signal and it needs to be handled.
- Show signal-int.c, run code and press ctrl+c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int quit = 0;

void handler (int signal) {
    printf("\nCaught signal %d\n", signal);
    quit = 1;

    //return;
    exit(-1); //to terminate
}

int main () {
    // SIGINT "program interrupt" signal, the signal is sent when the user type
    // INTR character, usually Ctrl+c
    signal(SIGINT, *handler);

    while (!quit) {
        printf("blah\n");
    }
    printf("\nCleaning up\n");

    return 0;
}
```

```
blah
blah
^Cblah
blah
blah
blah
blah
blah
Caught signal 2
```

- If the signal handler simply returns instead of exiting the program, the main program will continue where it left off. But it's not always possible. [comment out the exit(-1) in the above and remain return in the handler function, and run the code]
- Show signal-bus.c [if use return in the handler, it will generate infinite loop]
- Show signal-segv.c

```
#include
<stdio.h>
#include
```

```
Signal — -bash — 80x24 Signal — -bash — 80x24
mbp-190250:Signal yingli$ gcc signal-segv.c          signal-bus.c
mbp-190250:Signal yingli$ ./a.out                  .out
in do_something

Caught seg fault 11
mbp-190250:Signal yingli$
```

```
<stdlib.h>
#include <signal.h>
#include <string.h>

void bushandler( int signal ) {
    printf("Caught bus error %d\n", signal);
    // return; //infinite loop
    exit(-1); // gracefully leave program
}
```

```
int main(int argc, char *argv[]) {
    signal( SIGBUS, bushandler );
```

```
# GPACalc.py

def GPACalc (ary):
    sum = 0.0
    for f in ary:
        assert f >= 0.0 and f <= 4.3, "Invalid value " + str(f)
        sum += f;
    return sum/len(ary)

def main ():
    a = [3.5, 3.8, 4.0, 4.1, 4.5]
    print(GPACalc(a))

if __name__ == "__main__":
    main()
```

```
blah
^Cblah
blah
blah
blah
blah
blah
blah
blah
blah
Caught signal 2
Cleaning up
```

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

float *x = NULL;

// assume global variable x has been allocated. print its value.
void do_something() {
    printf( "in do_something\n" );
    // the error happens inside printf
    printf("%.2f\n", *x );
}

void seghandler(int signal) {
    printf("\nCaught seg fault %d\n", signal);
    exit(-1);
}

int main(int argc, char *argv[]) {
    signal(SIGSEGV, seghandler);
    do_something();

    return 0;
}
```