Programming Language Concepts

Programming Language Theory

Topics

- Control Structure
  - Expressions and Their Evaluation
  - Statement
  - **Control Flow and Recursion**
- Control Abstraction
  - Subprogram

Control Flow

- There are several kinds of control flow in programming language.
- Sequence
- Selection (or conditional)
- Iteration

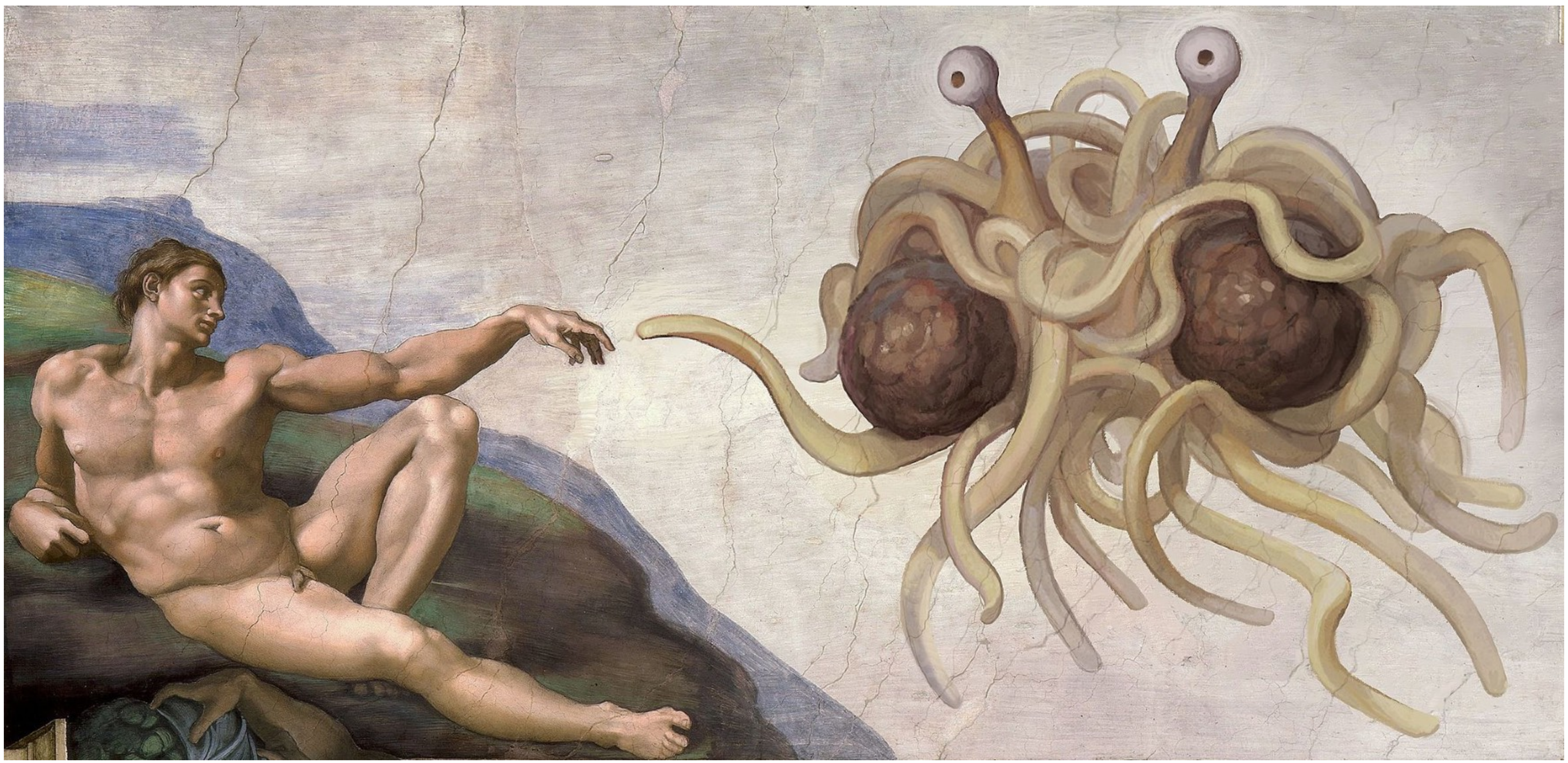
Sequence Control Statement

- Sequential and Composite
  - Sequence of statements often indicated as ";".
    - S1 ; S2 → *execution of S2 starts right after S1 terminates.*
  - We can group a sequence of statements into a Composite statement.
    - Usually using { } - code blocks.

Goto

- Similar to Jump instructions in assembly language.
- `goto Label`
- Immediately jump to the $Label$.
- If this statement is not used carefully, it easily generates a "*spaghetti code*".

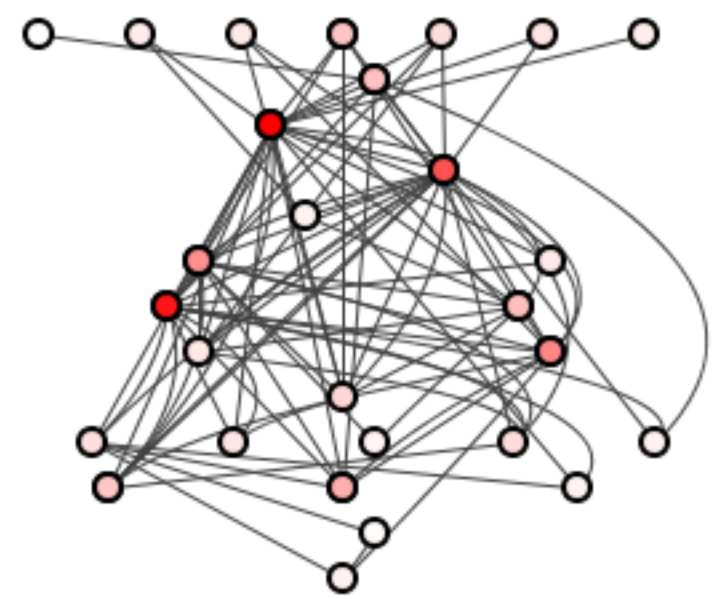
Flying Spaghetti Monster



Spaghetti Code

- If goto statement makes jumps to arbitrary locations in the code,
    - it is very difficult to trace the execution of a program.
- When we represent the connections made by the jumps,
    - It may look similar to spaghetti tangled in a plate.

```
• function a() {            • function b() {
      goto B;                     goto E;
      A: ....                     B: ....
          goto D;                     goto C;
  }                           }
```



From edmundkirwan blog

Demise of Goto

- In modern languages, goto statement is no longer popular.
- Also it is recommended not to use goto statements unless it is absolutely necessary.
- Most of its behavior can be supported by other statements in limited ways like `return`, `break` or `continue`.
- Although it has some historic value to know the evolution of programming languages, we will skip the details in this course.

Conditional Statements

- Evaluate a given boolean expression, and execute statements based on its value.

- Mostly it has a form like the following.

  - if <bool_expr> then C1 *else* C2

- Handling nested if statements.

  - if <bool_expr> then C1 *else* C2 *endif* **Using terminator**

  - if <bool_expr1> then C1
    else if <bool_expr2> then C2
    ....
    else Cn

    Using else if for nested ones.

Conditional Statements

- Case statement (or switch-case statement):
  - Handles many different branches.
  - Used when branches can be decided by an evaluation of an expression `<expr>`.
  - Each `<label>` represents a constant value or values.
- It is more efficient than if-else statements when there are many branches.

```
switch(<expr>) {
    case <label1>:
        S1;
        break;
    case <label2>:
        S2;
        break;
    ....
}
```

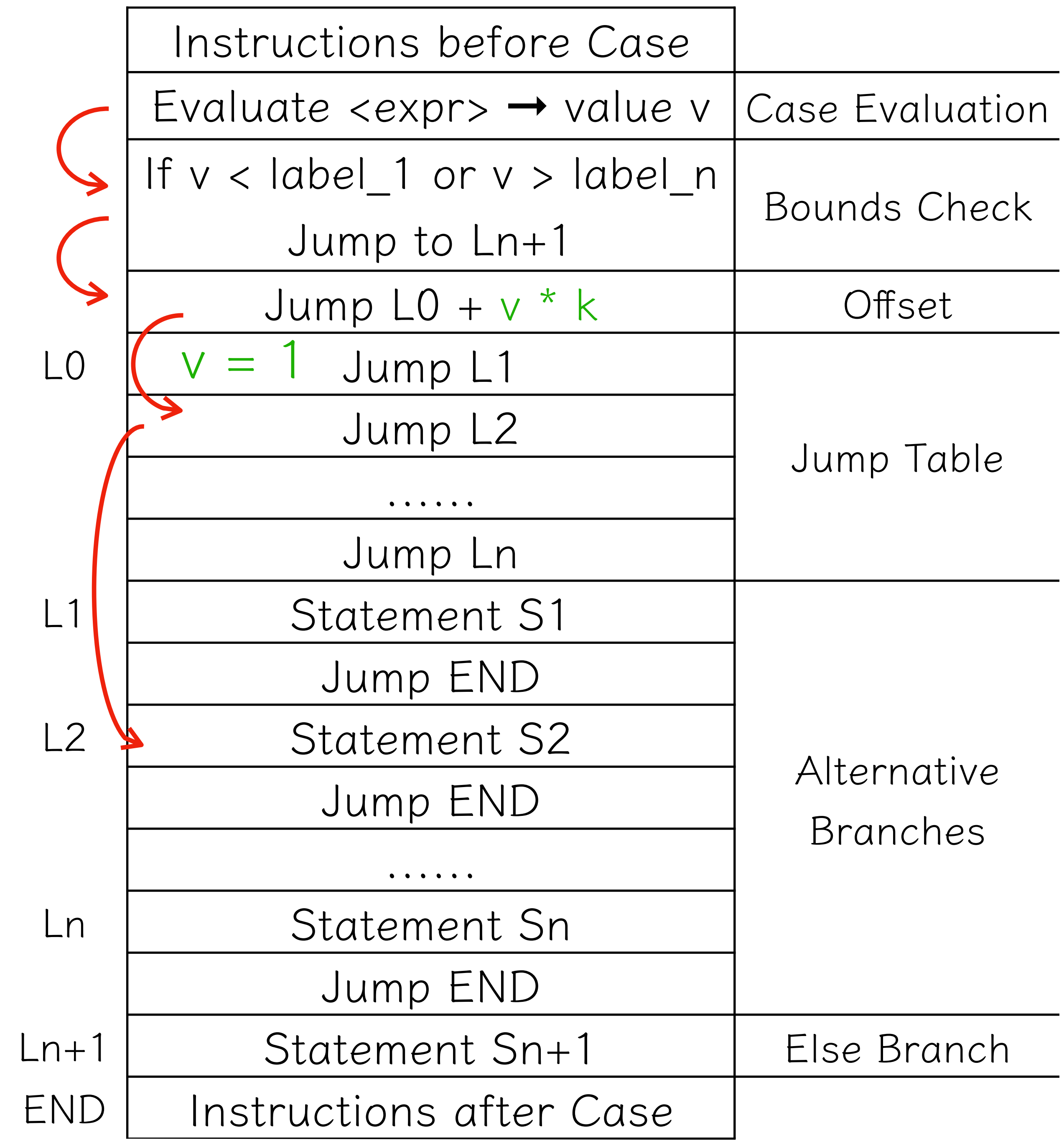
Implementation of Case

- We can implement case statement with a ***Jump Table***.
- Jump table contains jump instructions for each branch.
- After evaluate the value of `<expr>`, we can get *an offset* for the jump table.

	Instructions before Case	
	Evaluate <expr> → value v	Case Evaluation
	If v < label_1 or v > label_n Jump to Ln+1	Bounds Check
	Jump L0 + v * k	Offset
L0	v = 1 Jump L1	Jump Table
	Jump L2	
	......	
	Jump Ln	
L1	Statement S1	Alternative Branches
	Jump END	
L2	Statement S2	
	Jump END	
	......	
Ln	Statement Sn	
	Jump END	
Ln+1	Statement Sn+1	Else Branch
END	Instructions after Case	

Implementation of Case

- Without Jump End (break), the execution will continue to the next.
- With this implementation, we need a large jump table if constants of label ranges widely.
    - e.g.) two cases 0 and 1000.
    - We need to have 1~999 in Jump Table although they are not used.
    - We can calculate jump address in different ways such as hashing.



Iterations

- Iterative statements can be distinguished in two major categories:
  - Unbounded iteration
    - Often implemented as `while` statements.
  - Bounded iteration
    - Often implemented as `for` statements.
- Employing iterations gives the expressive powers so that a language can be Turing complete.
  - We can write all computable algorithms with this language.

Unbounded Iteration

- Unbounded iteration is implemented by two parts:
    - a loop ***condition*** and a loop ***body***.
    - ```
      while <bool_expr>  ⟵  condition
      do
          <statement>  ⟵  body
      ```
- Repeats the execution of the body while the condition is satisfied (i.e. evaluated as true).
      ```

Bounded Iteration

- Bounded iteration is implemented with more complex components.

- ```
  for i = <start> to <end> by step
  do
      <statement>  ←—— body
  ```

  - It usually has a variable i called the *index*, or *counter* or *control variable*.

  - Then it modifies the variable by `step`, which is a non-zero integer constant.

  - `<start>` and `<end>` are expressions for range.
  ```

Unbounded vs. Bounded

- We cannot see many "pure" bounded iteration statements.
- For bounded iteration, at the start of iterations, we can know ***the number of iteration***.
- This is not the case for unbounded iteration.
- e.g.) In C, for statement is not pure bounded iteration.
  - ```
    for (int i=0; i<n; i++) {
          ...
          n += 1;
    }
    ```

    updating the condition is possible in the body of the loop.
    ```

Recursion

- A function or a procedure is called ***recursive***, if it calls itself inside its body.

- Recursion is another mechanism to obtain Turing completeness.

- Recursion is appeared commonly in mathematics, which is often called inductive definition.

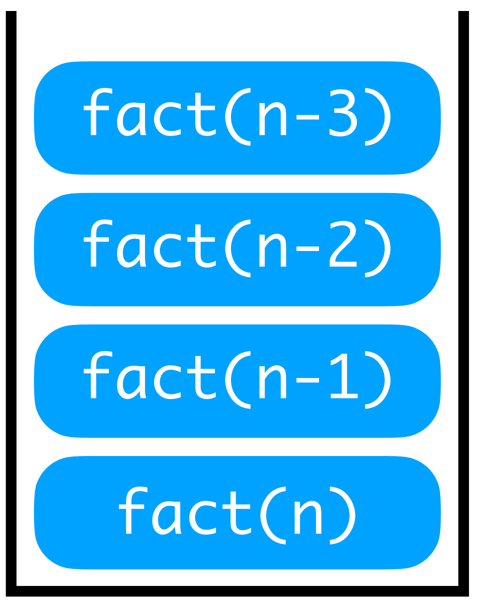
  - factorial(n) = 1 n = 1

    n*factorial(n-1) otherwise

Recursion in PL

- Recursion is often considered inefficient compared to iteration.
- Because it continuously calls itself over and over.
- For each call, we have to push a new activation record into the stack, to store parameters and return values.

```
int fact(int n) {
    if(n == 1)
        return 1;
    else
        return n*fact(n-1);
}
```



Tail Recursion

- It would be much more efficient, if we share the activation records for each recursive call.
- It is possible with ***tail recursion***, which returns a return value without any additional computations.
- We may introduce a new variable to store intermediate results as parameters.

Recursion

```
int fact(int n) {
    if(n == 1)
        return 1;
    else
        return n*fact(n-1);
}
```

Tail Recursion

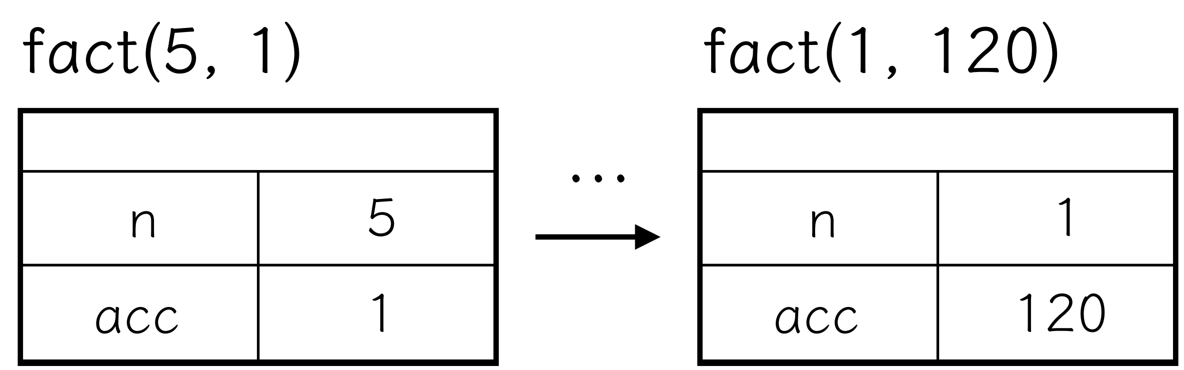
```
int fact(int n, int acc) {
    if(n == 1)
        return acc;
    else
        return fact(n-1, n*acc);
}
```

Directly returns parameters or return values

Tail Recursion

- It would be much more efficient, if we share the activation records for each recursive call.
- It is possible with ***tail recursion***, which only returns the return value of its recursive call, without any extra computation.
- We may introduce a new variable to store intermediate results as parameters.

```
int fact(int n, int acc) {
    if(n == 1)
        return acc;
    else
        return fact(n-1, n*acc);
}
```



Single Activation Record

Summary

- Control Flow
  - Sequence and Composite
  - Conditional
  - Iteration
- Recursion and Tail Recursion