Control Structure

Programming Language Theory

Topics

- Control Structure
  - Expressions and Their Evaluation
  - Statement
  - Control Flow and Recursion

Expressions

Expression

- An ***Expression*** is a syntactic entity whose evaluation either ***produces a value or undefined*** (which fails to terminate).

- Expressions are one of the basic components of every programming language.

- Although there are languages such as functional languages which do not have statements, expressions exist in every language.

How to Represent?

- Operator and Operands.
  - `x + y`, `b - 1`, `f(3) >= 0`
- Prefix, Infix, Postfix notations.
  - Based on the location of operators,
  - `<prefix> ::= <op><prefix><prefix>` | ...
  - `<infix> ::= <infix><op><infix>` | ...
  - `<postfix> ::= <postfix><postfix><op>` | ...

Notations

- Consider mathematical equation: $a + b * c + d$
- Infix Notation: $(a + b) * (c + d)$
- Prefix Notation: $* + a\ b + c\ d$ - Also called *prefix Polish* notation.
    - or $(* \ (+ \ a \ b) \ (+ \ c \ d))$ - *Cambridge Polish* notation, puts operators inside parentheses.
- Postfix Notation
    - $a\ b + c\ d + *$

Semantics

- The semantics of expressions (or how they are evaluated) can be changed according to notations.

- For instance, infix expressions without parentheses may cause ambiguity in its evaluation.

  - $a + b * c + d$

  - $a + (b * c) + d$? or $(a + b) * (b + c)$?

- With *Infix Notation*, we need to consider ***Precedence*** and ***Associativity*** of operators.

Precedence

- Operator Precedence decides which operators should be considered first.
- We need to define such precedence to make evaluation of expression match to our intuition.
  - For `1 + 2 * 3`, we want its value to be 7, not 9.
    - `1 + (2 * 3) = 7 vs. (1 + 2) * 3 = 9`
- So we need precedence rules to prevent such cases.

Associativity

- However, precedence is not enough to correctly evaluate expressions.
- We also need to consider ***operator associativity***, tells us how an operator associates with its operands.
  - `10 - 5 - 3`
    - `(10 - 5) - 3 = 2 vs. 10 - (5 - 3) = 8`
- Most of arithmetic operators associate from ***left to right***,
  - but there is a case like exponentiation,
  - 5^{3^2}= `5^3^2` → `5^(3^2) vs. (5^3)^2`

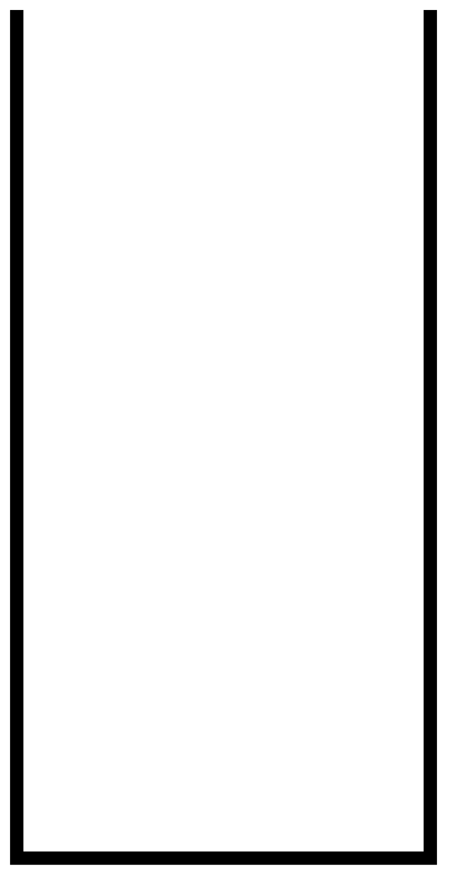
Precedence and Associativity

- Most languages have intuitive precedence and associativity.
- We need to carefully consider them when writing code.
- If you have any suspicion, use parentheses to clarify your intention.
  - `(1 + 2) * 3, (10 - 5) - 3, (5^3)^2`

Prefix Notation

- Unlike infix notations, prefix notation has no such ambiguity, if we know the ***arity*** *(# of operands) of an operator.*

- We can consider a simple algorithm to evaluate prefix expressions with a stack and a counter.
    - $* + a\ 2 + b\ c$
    - $a = 1,\ b = 2,\ c = 3$



Input: $* + a\ 2 + b\ c$

Counter C = 0

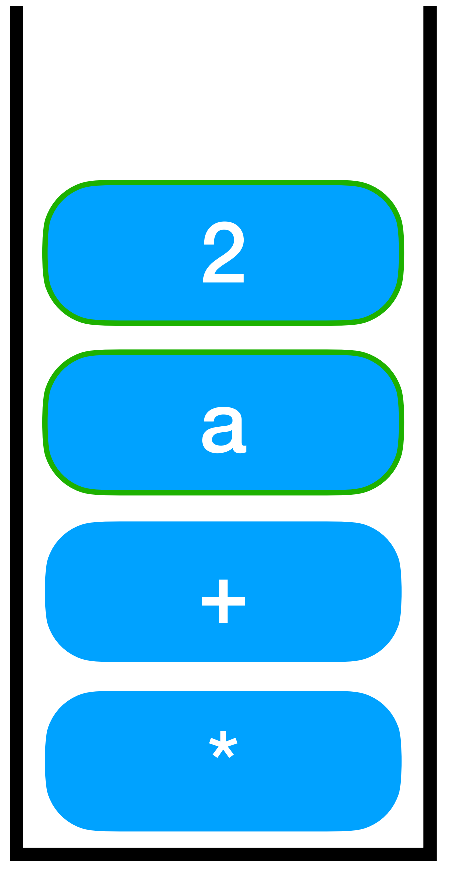
Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
    - If it is an operator, update C with the arity.
    - Each operand symbol, decrease C.
    - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
    - Update C for new operator.

a = 1, b = 2, c = 3

Input: * + a 2 + b c

Evaluate!



stack

Counter C = 0

Counter C = 1

Counter C = 2

Counter C = 2

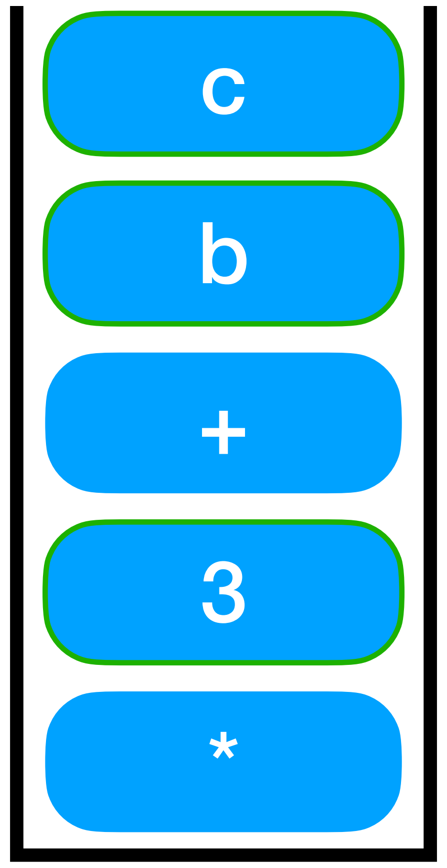
Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
    - If it is an operator, update C with the arity.
    - Each operand symbol, decrease C.
    - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
    - Update C for new operator.

a = 1, b = 2, c = 3

Input: * + a 2 + b c

Evaluate!



Counter C = 0

Counter C = 1

Counter C = 2

Counter C = 1

stack

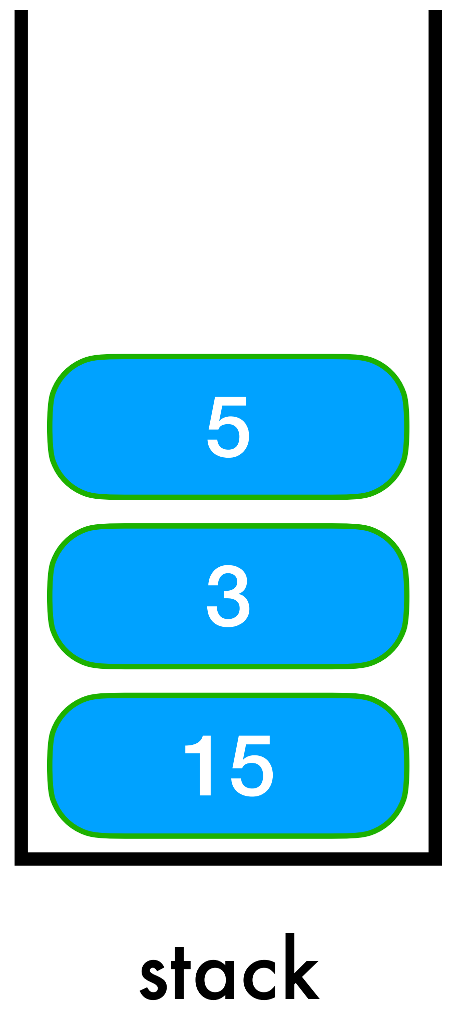
Prefix Notation

- Counter C = 0.
- Push each symbol to a stack.
    - If it is an operator, update C with the arity.
    - Each operand symbol, decrease C.
    - If C = 0, apply operator and store the result R to the stack, then delete evaluated symbols.
    - Update C for new operator.

a = 1, b = 2, c = 3

Input: * + a 2 + b c

Evaluate!



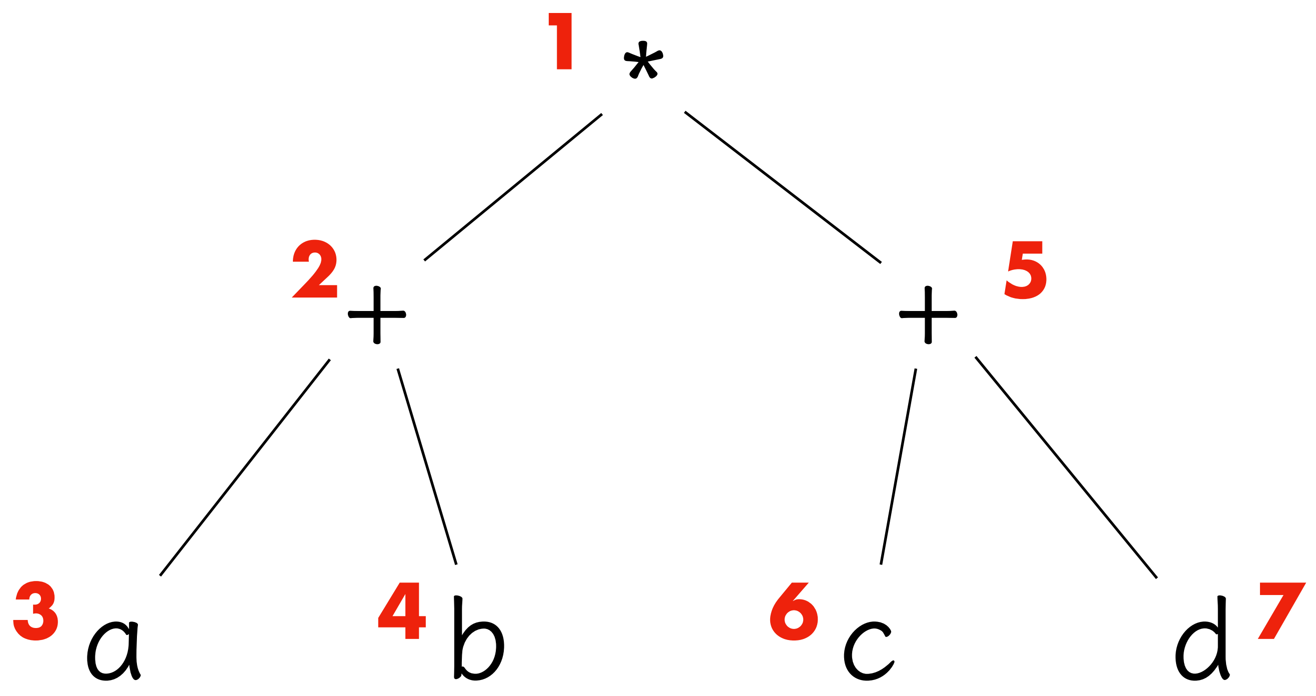
Counter C = 0

Postfix Notation

- In postfix notation, it is even simpler.
- We can read symbols from left to right, and every time we meet an operator, apply it to previous symbols based on its arity.
  - $a\ b + c\ d + *,\ a = 1,\ b = 2,\ c = 3,\ d = 4$
  - $\underline{a\ b\ +}\ c\ d + * \rightarrow 3\ \underline{c\ d\ +}\ * \rightarrow \underline{3\ 7\ *} \rightarrow 21$

Using Syntax Tree

- We can also parse an expression into a syntax tree, then consider it with different traversal orders.
  - Non-leaf nodes are operators,
  - leaf nodes are operands.
- a + b * c + d



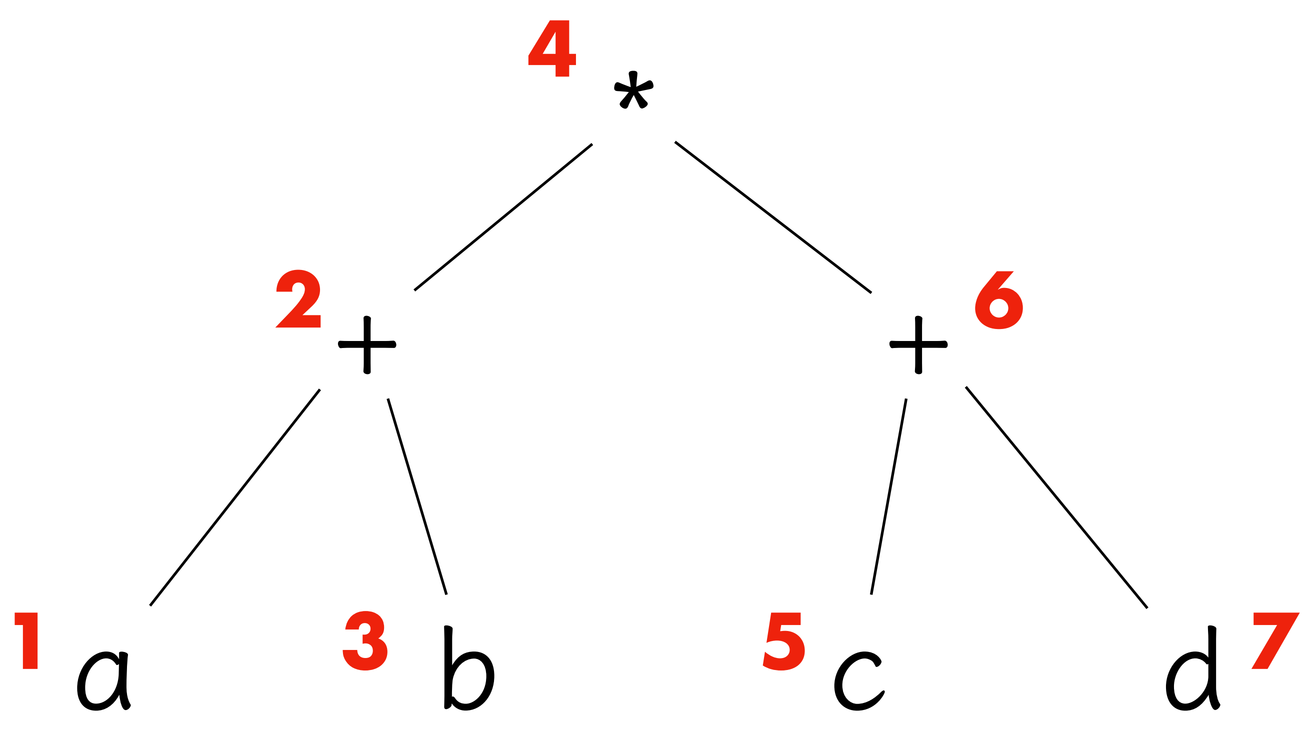
Pre-order * + a b + c d

In-order

Post-order

Using Syntax Tree

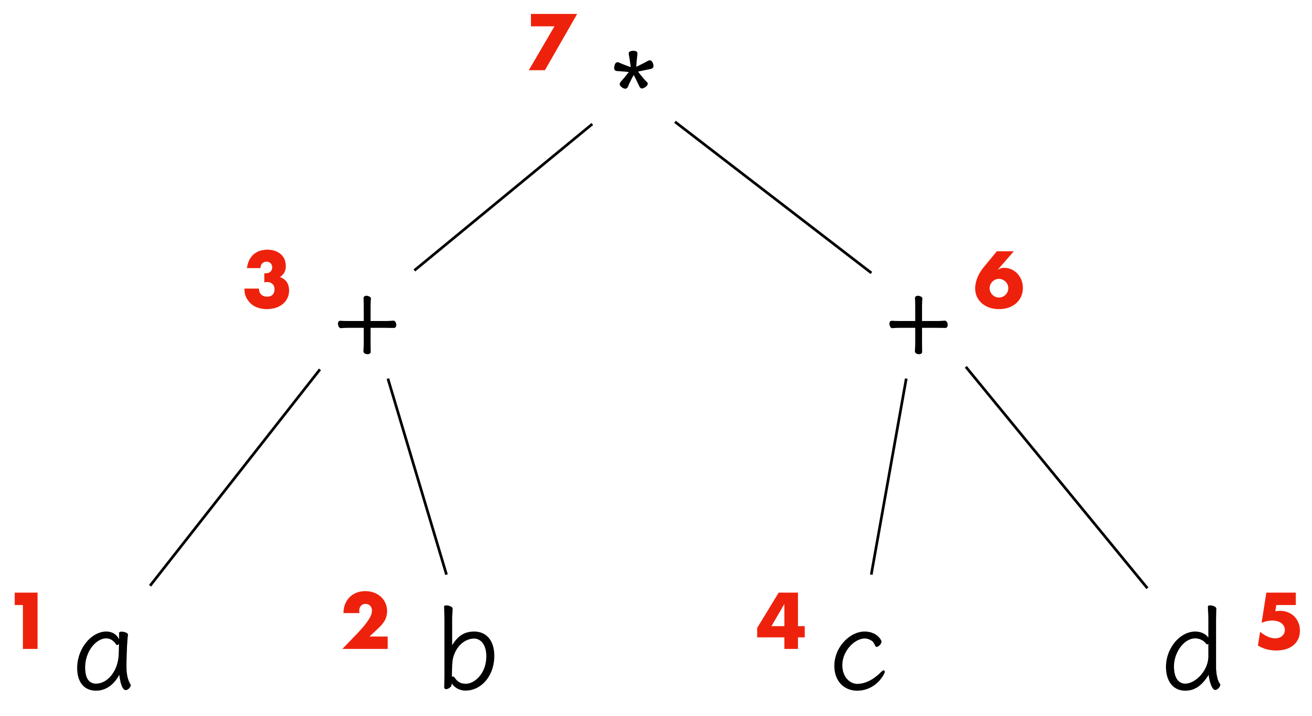
- We can also parse an expression into a syntax tree, then consider it with different traversal orders.
  - Non-leaf nodes are operators,
  - leaf nodes are operands.
- a + b * c + d



Pre-order	* + a b + c d
In-order	a + b * c + d
Post-order	

Using Syntax Tree

- We can also parse an expression into a syntax tree, then consider it with different traversal orders.
  - Non-leaf nodes are operators,
  - leaf nodes are operands.
- $a + b * c + d$



Pre-order	* + a b + c d
In-order	a + b * c + d
Post-order	a b + c d + *

Expression Evaluation

- In mathematics, a - b + c and a + c - b do not have different results - they are mathematically equivalent.
- However, in PL expressions, such ***Subexpression Evaluation Order*** can actually modify the result.
- Hence we have to consider subexpression evaluation order.
- There are several reasons why we should be careful.

Side Effect

- In imperative languages, it is possible that evaluation itself modifies the value of a variable through side effect.

- (a + b++) * (c + b--)

- (a + f(b)) * (c + f(d))

- A component of a program has a side effect if it modifies the state of a program by execution.

Finite Arithmetic

- Numbers represented in a computer are ***finite***.
- e.g.) In C, we have different integer types such as `short`, `int`, and `long`, which can represent different range of integers.
- If the result of a computation (or evaluation of a subexpression) exceeds the boundaries, there will be overflow or underflow.
  - $a-b+c \rightarrow (a-b)+c$ vs. $(a+c)-b$, $b > c$
  - In computer there might be a problem for the latter, if $(a+c)$ is out of range, while the former can be OK due to $(a-b)$.

Undefined Operands

- Two strategies of Operator Application: ***eager evaluation*** or ***lazy evaluation***.
- *Eager evaluation* first computes all subexpressions, then apply operators.
- *Lazy evaluation* decides the evaluation of a subexpression later.
    - `a == 0 ? b : b/a` → "b over a" means "a divided by b".
    - If we evaluate all the operands first, it will cause an error while evaluating `b/a`, since ***divide by 0 is undefined***.
    - But it is okay if we only evaluate an operand which need to be evaluated - `a == 0`, then `b` or `a != 0` then `b/a`.

Short-circuiting

- Short-circuiting is a technique to only evaluate a partial expression when the other is not required to be evaluated.
  - `if(str != null && str.length() > 0) ...`
  - If `str != null` is not satisfied, we don't need to evaluate `str.length()`.
  - Actually, evaluating `str.length()` before `str != null` will cause a problem.

Code Optimization

- The subexpression evaluation order may affect the efficiency of evaluation itself, considering code optimization.
    - ```
      a = array[i];
      b = a*a + c/d;
      ```
    - As you may already know, value of `a` should be read from the memory.
    - Hence it might be more efficient to evaluate `c/d` first.
      ```

Statements

Statement

- A *Statement* is a syntactic entity whose evaluation doesn't necessarily return a value, but can have a side effect.

- *Statements* are not present in all programming languages, but they are typically used by *Imperative Languages*.

- By executing (or evaluating) statements, we can keep changing a program's state.

    - e.g.) `print("Hello World!")`

Ambiguity in Definition

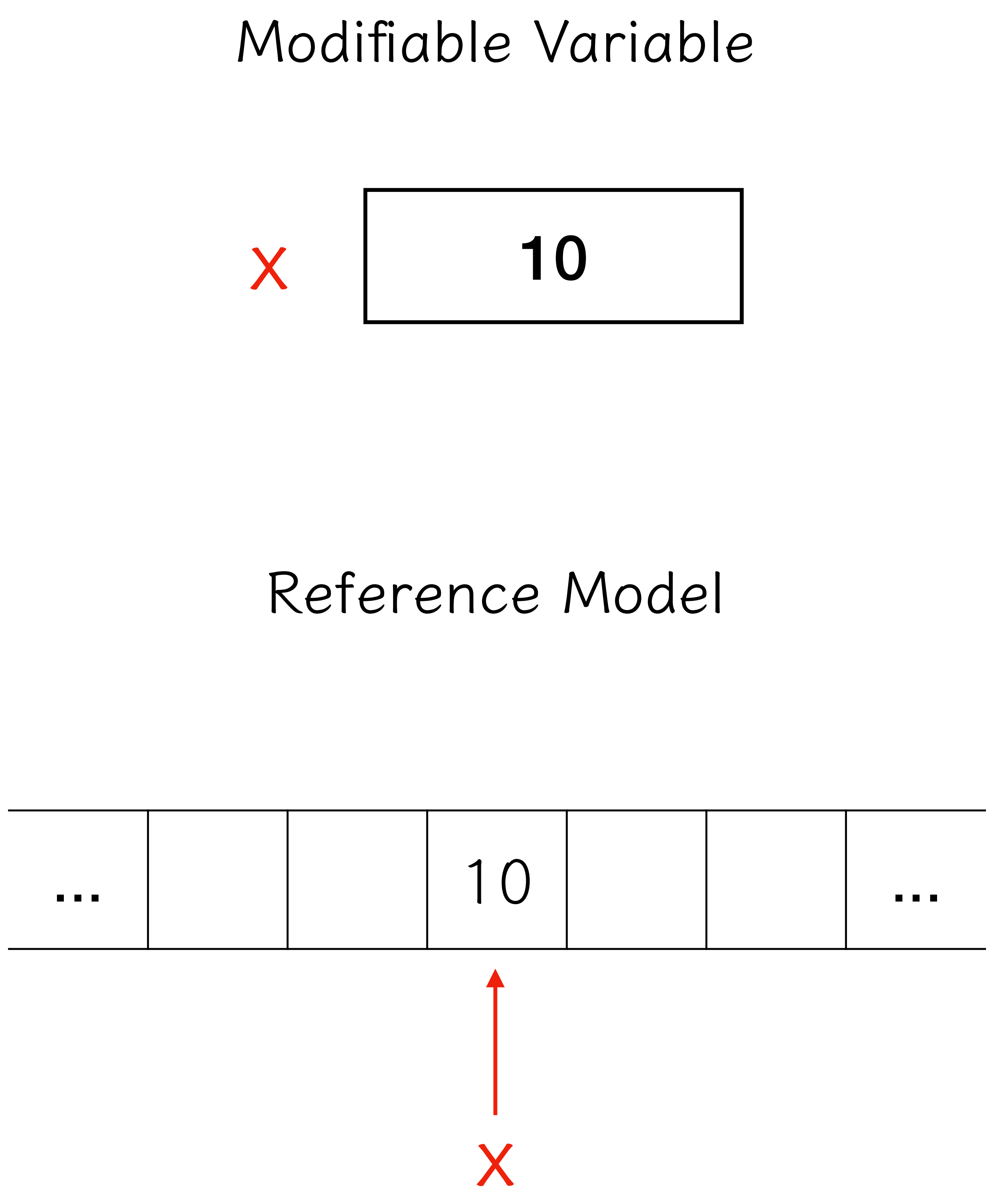
- We used the term "evaluation", which is not precisely and exactly defined, to define expression and statement.
- In different languages, an expression may have a side-effect, and a statement can have a return value.
  - In C, an assignment modifies the value of a variable, as well as returns the value.
- The key distinction is that when the state is fixed before the evaluation,
  - the result of expression evaluation is ***a value,***
  - while the result of statement evaluation is ***change of the state***.

The Concept of Variable

- In programming languages, two models of variables are employed.
- Modifiable Variable
  - A variable is considered as a container or location, which stores a value.
  - The value is "***modifiable***", by executing assignments.
- Reference Model
  - A variable is considered as a reference to a value stored in the memory, not a container of a value.

The Concept of Variable

- In modifiable variable, a variable itself is a container.
- In reference model, a variable is merely a reference to a memory location.
- Note that this is the concept of variable, and its implementation can be different in each language.



Assignment

- ***Assignment*** is a statement which modifies a value associated with a modifiable variable.

- `<assign> ::= <expr1><opAssign><expr2>`

- For `<expr1>`, we use the ***l-value***, and for `<expr2>` we need the ***r-value***.

  - `x = 3; x = x + 1;`

  - On the left side, we use *l-value of* `x` *(the location)*, and on the right side, we use *r-value of* `x` *(value 3)*.

Assignment

- How assignment works with a variable of reference model?
  - `x = y`
  - It doesn't mean copying the value of `y` to variable `x`.
  - Rather they are now ***two references to the same object***.
    - We can modify `y` and it can be seen via `x`.
    - Similar to pointer variables, but in reference model, we can only modify the value indirectly with assignments.
- Java is a language employs reference model for variables of class types.

Control Flow and Recursion

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.

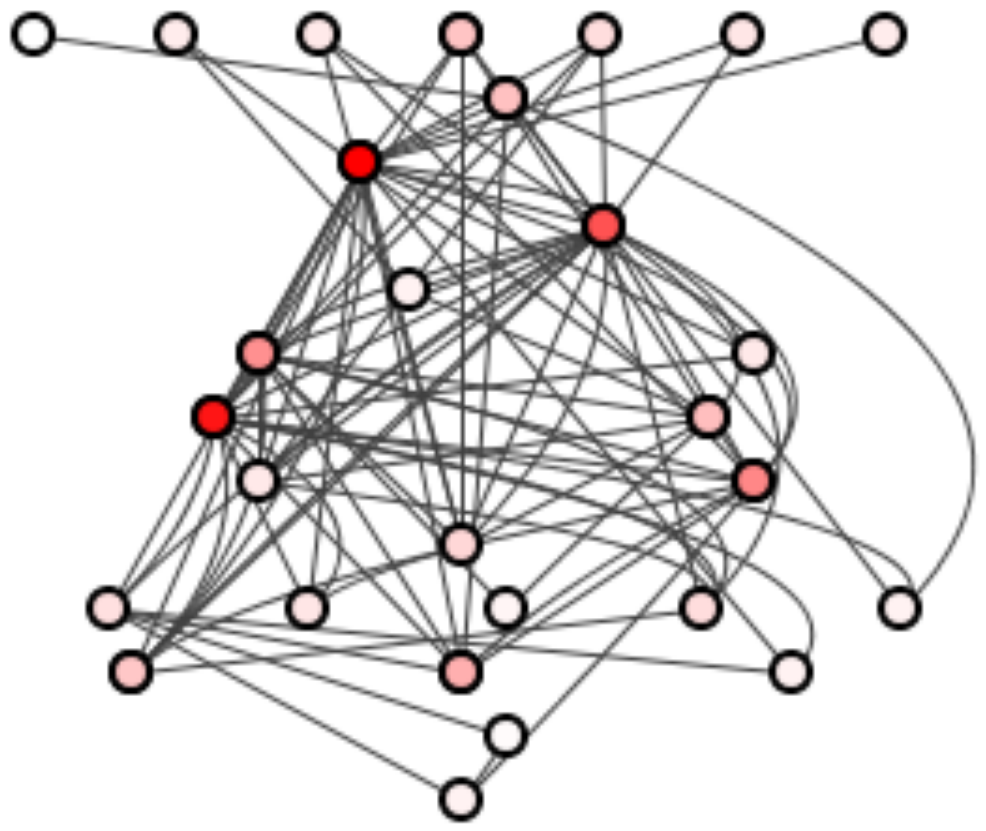
Goto

- 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*".

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() {
      goto B;
      A: ....
          goto D;
  }
  ```
- ```
  function b() {
      goto E;
      B: ....
          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 **Using else if for nested ones.**
    ....
    else Cn

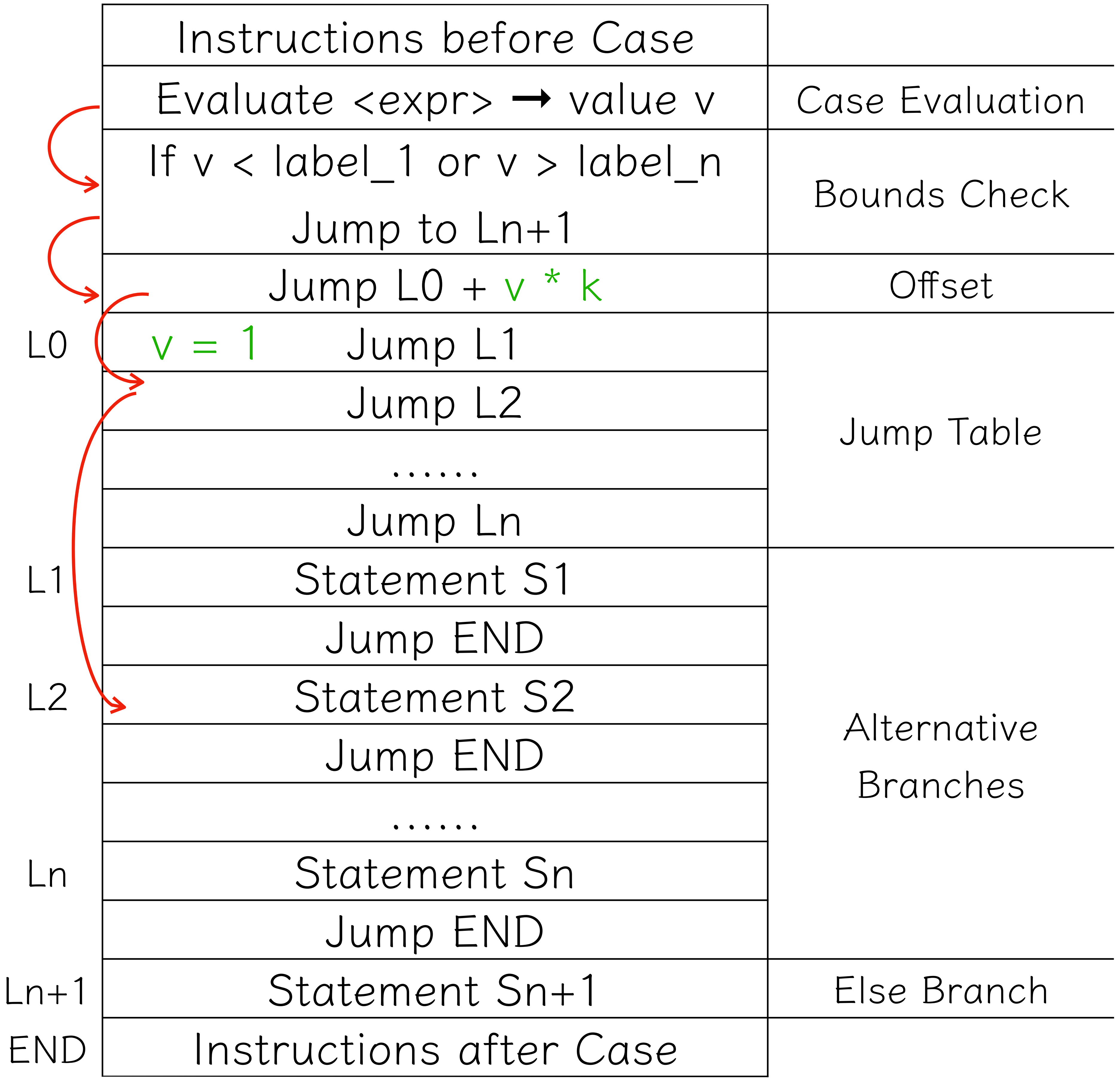
Conditional Statements

- 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.



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.

	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	

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 iterations***.
- This is not the case for unbounded iteration.
- e.g.) In C, for statement is not pure bounded iteration.
  - ```c
    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.

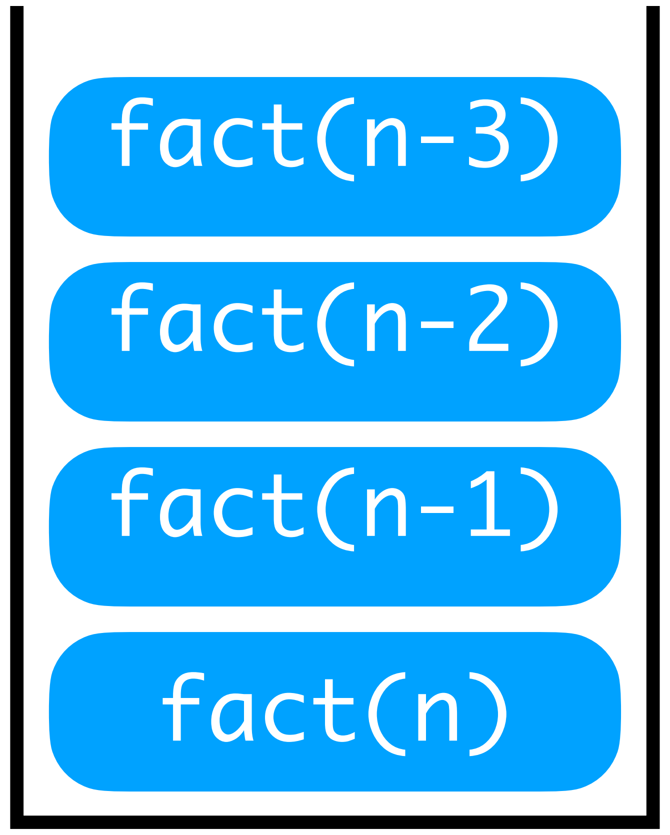
- Recursion is appeared commonly in mathematics, which is often called inductive definition.

  - 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 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.

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);
}
```

fact(5, 1)

n	5
acc	1

… →

fact(1, 120)

n	1
acc	120

Single Activation Record

Summary

- Expressions and Their Evaluation
- Statement
- Control Flow - Conditional, Iteration
- Tail Recursion