Welcome

CS1101S Studio Session Week 4:
Higher-order Programming & Language Processing

Niu Yunpeng

niuyunpeng@u.nus.edu

September 4, 2018

Before We Start

Reading Assessment 1

- Coming soon in Week 4
- Focus on many topics until Week 3
    - Basic programming
    - Substitution model
    - Recursion & iteration
    - Scoping
    - ...

Good luck!

- Try to get full marks.

Before We Start

`if-else` block

- Introduced in Week 3 lecture
- A very important building block in larger programs
- Many sections in Week 3 studio slides should be readable to you now
- ...

Overview

1. More about recursion
   - From last week
   - Wishful thinking
   - Examples
2. Higher-order programming
   - Before we start
   - To understand higher-order programming
   - To use higher-order programming
   - Exercises
3. Language processing
   - Family of programming languages
   - From low-level to high-level
   - Compilation & interpretation

Review

A few terms so far

- Primitives/combination/abstraction
- Recursive/iterative function
- Recursive/iterative process

Review

Two approaches

We have two general approaches to solve a really large problem:

- Bottom-up approach: begin with all the smallest units of this problem and combine them together.
- Top-down approach: repeatedly divide a larger problem into several smaller problems and "**wish**" these sub-problems could be solved.

Two programming styles

- Iteration: the bottom-up approach;
- Recursion: the top-down approach.

Review

To understand recursion

- Use ***substitution model*** (*applicative order reduction*).

Substitution model

To use substitution model on understanding a function:

- Evaluate all actual arguments;
- Replace all formal parameters with their actual arguments;
- Apply each statement in the function body (and get the return value);
- Repeat the first 3 steps until done.

Wishful Thinking

What is wishful thinking?

Explained in the textbook, *Structure and Interpretation of Computer Programs* (click *here* to read).

Interpretation

- Why the recursive calls could solve the sub-problems?
- Because I "**wish**" those sub-problems could be solved.
- Thus, I just need to consider how to combine them together.

Wishful Thinking

More Recursion

Classical examples of recursion

- Factorial
- Square root
- Power function
- Fibonacci
- Greatest common divisor (GCD)
- Least common multiple (LCM)
- Hanoi tower
- Coin change
- Permutation / combination
- ...

More Recursion

Examples in Week 3 slides

- Factorial
- Square root
- Power function
- Fibonacci
- Greatest common divisor (GCD)
- Least common multiple (LCM)

More Recursion

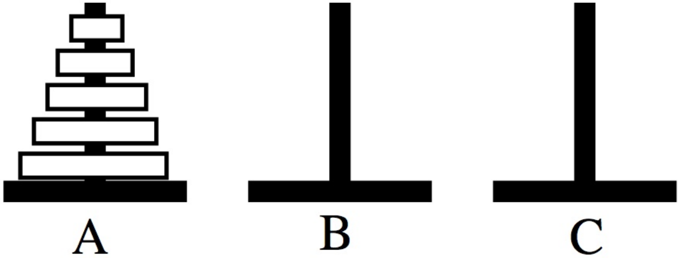
Examples in Week 4 slides

- Hanoi tower
- Coin change

More Recursion

Hanoi tower

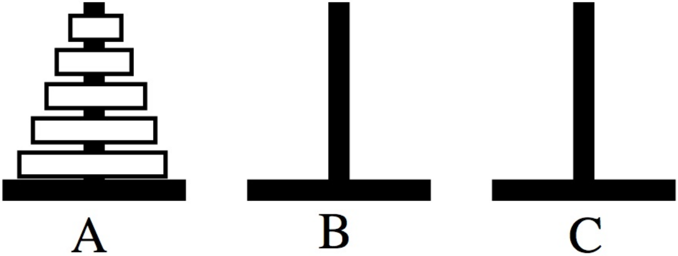
- Given: a tower consisting of disks in increasing size;
- Goal: move all disks from A to B with the help of C;
- Constraint: never put a larger disk on top of a smaller one.



More Recursion

Recursion for Hanoi tower

- Base case: move 2 disks from A to B with the help of C.
- Scale: n disks.
- Sub-problem: how to solve the problems of $n-1$ disks.



More Recursion

Hanoi tower

```
function hanoi(size, from, to, extra) {
    if (size === 0) {
        ;
    } else {
        hanoi(size - 1, from, extra, to);
        move_disk(from, to);
        hanoi(size - 1, extra, to, from);
    }
}
```

More Recursion

An interesting concern

- What is the `move_disk` function?
- Where is it defined?
- Why do we need it?

Answer

- It does not matter. It is simply an abstraction.
- It is just a way to tell you that, the top disk will be moved from somewhere to elsewhere.

More Recursion

Coin change

- Given: a set of unlimited coins (however limited number of kinds);
- Given also: a specific amount of money in cents;
- Goal: find the number of ways to change this amount into coins.

More Recursion

Recursion for coin change

- Base case: the amount of money left is 0, which means a valid way to make the changes.
- Scale: the amount of money left *in cents*.
- Sub-problem: to use the same kind or a new kind.

More Recursion

Recursion for coin change

- Base case: the amount of money left is 0, which means a valid way to make the changes.
- Scale: the amount of money left *in cents*.
- Sub-problem: to use the same kind or a new kind.

More Recursion

Coin change

```
function coin_change(amount, kind) {
    if (amount === 0) {
        return 1;
    } else if (amount < 0 || kind === 0) {
        return 0;
    } else {
        return coin_change(amount, kind - 1) +
               coin_change(amount - value(kind), kind);
    }
}
```

More Recursion

Coin change

```
function value(kind) {
    return kind === 1 ?   5 :
           kind === 2 ?  10 :
           kind === 3 ?  20 :
           kind === 4 ?  50 :
           kind === 5 ? 100 :
           0;
}
```

More Recursion

What is coin change really about?

- It is to count the number of ways we can solve a problem.
- In fact, it is to count the number of leaves in a *decision tree*.

More Recursion

What is coin change really about?

- It is to count the number of ways we can solve a problem.
- In fact, it is to count the number of leaves in a *decision tree*.

What?

- Unbelievable! We are learning part of the simplest form of *machine learning (ML)* or *artificial intelligence (AI)*.

More Recursion

AlphaGo vs Lee Sedol two year ago

More Recursion

Recommended modules at SoC

- CS3243(R) Introduction to Artificial Intelligence
- CS3244 Machine Learning
- CS4246 AI Planning and Decision Making
- CS5339 Theory and Algorithms for Machine Learning
- CS5340 Uncertainty Modelling in AI

Caution

- Hard modules;
- Need strong mathematical foundations.

More Recursion

Examples we have learn so far...

- Factorial
- Square root
- Power function
- Fibonacci
- Greatest common divisor (GCD)
- Least common multiple (LCM)
- Hanoi tower
- Coin change

One thing left...

- Permutation / combination

Overview

1. More about recursion
   - From last week
   - Wishful thinking
   - Examples
2. Higher-order programming
   - Before we start
   - To understand higher-order programming
   - To use higher-order programming
   - Exercises
3. Language processing
   - Family of programming languages
   - From low-level to high-level
   - Compilation & interpretation

Higher-order Programming

Before we start...

We need to mention a few things before we start:

- How to check the correctness of a program;
- Revisit of variable scoping;
- Why we can do higher-order programming in JavaScript?

Higher-order Programming

How to check the correctness of a program

- Invariant
- Termination
    - Base case(s)
    - Finite time/space complexity

Higher-order Programming

Order of growth exercise from last week

```
function d(n) {
    if (n < 0) {
        return 0;
    } else {
        return d(n / 3);
    }
}

d(10);
```

Question

- Will it terminate?

Higher-order Programming

Revisit of variable scoping

- **Pre-defined** functions or constants are visible everywhere.
- A function or constant is visible within the closest surrounding curly braces where it is declared. Or it will be visible in the whole program if none (top-level constants, **global constants**).
- **Formal parameters** are visible within the function body to which it belongs.

Higher-order Programming

Core built-in functions

- `alert`
- `display`
- `error`
- `prompt`
- `parse_int`
- `runtime`

A few keywords

- `undefined`
- `Infinity`
- `-Infinity`
- `NaN`

Higher-order Programming

Mathematical library - functions

- `math_abs(x)`
- `math_sin(x) math_cos(x) math_tan(x)`
- `math_asin(x) math_acos(x) math_atan(x) math_atan2(y, x)`
- `math_floor(x) math_ceil(x) math_round(x)`
- `math_max(a, b, ...)  math_min(a, b, c, ...)`
- `math_pow(x, y) math_exp(x)`
- `math_sqrt(x)`
- `math_log(x) math_log10(x) math_log2(x)`

Higher-order Programming

Mathematical library - constants

- `math_E`
- `math_PI`
- `math_SQRT2`
- `math_SQRT1_2`
- `math_LN10`
- `math_LN2`

Higher-order Programming

Exercises for variable scoping

- Find out the output of each program, and
- Explain the reason.

Higher-order Programming

Exercise 1

```
const x = 5;

function f(x) {
    return x;
}

f(3);
```

Higher-order Programming

Exercise 2

```
const x = 5;

function f(x) {
    function g() {
        return x;
    }

    return g();
}

f(x);
```

Higher-order Programming

Things...

- Constants can be functions.
- Parameters can be functions.
- Return values can be functions.

Result...

- That's all about higher-order programming.

Higher-order Programming

Arrow function

- A more concise way to declare functions
- Especially useful for those one-line functions

Example

```
const circle_area = radius => math_PI * radius * radius;
circle_area(3);
```

Higher-order Programming

Original version

```
function fact(n) {
    // By definition, the factorial of 0 is 1.
    return n === 0 ? 1 : fact(n - 1) * n;
}
```

Notice

- This version gives rise to a recursive process.

Higher-order Programming

Abstract the multiplication

```
function make_multiplier(x) {
    return num => num * x;
}

const multiply_by_4 = make_multiplier(4);
multiply_by_4(5);
```

Higher-order Programming

Using the abstraction of multiplication

```
function fact(n) {
    if (n === 0) {
        return 1;
    } else {
        return (make_multiplier(n))(fact(n - 1));
    }
}
```

Higher-order Programming

Abstract the sub-problem relationship

```
function product(value, next, upper, lower) {
    if (upper <= lower) {
        return 1;
    } else {
        return value(upper) *
               product(value, next, next(upper), lower);
    }
}
```

Higher-order Programming

Abstract the relationship again

```
function product(value, next, terminate, now) {
    if (terminate(now)) {
        return 1;
    } else {
        return value(now) *
               product(value, next, terminate, next(now));
    }
}
```

Higher-order Programming

Think about it carefully...

Three key aspects for a recursive function:

- Base case(s)
- Scale
- Sub-problem(s)

Three functions as parameters for `product`:

- `terminate`
- `value`
- `next`

Higher-order Programming

Using the abstraction for sub-problem relationship

```
function fact(n) {
    return product(x => x,
                   x => x - 1,
                   x => x <= 0,
                   n);
}
```

Higher-order Programming

What about this?

- $1 + 2 + \cdots + n$
- $1 \times 2 \times \cdots \times n$
- For these two different series, what is in common?

Higher-order Programming

Abstract the multiplication and sub-problem relationship

```
function accum(value, next, terminate, operation, now) {
    if (terminate(now)) {
        return 1;
    } else {
        return operation(value(now),
                         accum(value, next, terminate,
                               operation, next(now)));
    }
}
```

Higher-order Programming

Once again

```
function accum(value, next, terminate, oper, base, now) {
    if (terminate(now)) {
        return base();
    } else {
        return oper(value(now),
                        accum(value, next, terminate, oper,
                              base, next(now)));
    }
}
```

Think about it...

- What changes?

Higher-order Programming

Using everything together

```
function fact(n) {
    return accum(x => x,
                 x => x - 1,
                 x => x <= 0,
                 (x, y) =>  x * y,
                 () => 1,
                 n);
}
```

Think about it...

- What changes?

Higher-order Programming

Your task today...

- Does this function gives rise to a recursive or iterative process?
- If it gives rise to a recursive process, can you change it into an iterative process?

Higher-order Programming

Notice

- In the following slides, you are going to see a few problems.
- They are selected from past year papers.

Higher-order Programming

Exercise 1

See the function `strict` below. Consider a restricted version of Source, in which each function is only allowed to have at most 1 parameter.
Find out how to achieve the same result as `strict` under this constraint.

```
function strict(a, b, c) {
    return a * b + c;
}
```

Higher-order Programming

Exercise 2

```
function plus_one(x) {
    return x + 1;
}

function trans(func) {
    return x => 2 * func(x * 2);
}

function twice(func) {
    return x => func(func(x));
}
```

Higher-order Programming

Exercise 2

Given the three functions in the last page, try to find out the output of the following programs:

- `((twice(trans))(plus_one))(1);`
- `((twice(trans(plus_one))))(1);`

Higher-order Programming

Exercise 3

- According to the substitution model of execution, a process can be said to *exhaust all time resources* if it keeps evaluating and never reaches any result value.
- Also, a process can be said to exhaust all space resources if it keeps growing while it evaluates sub-expressions, i.e. the number of sub-expressions and deferred operations will keep growing.

Higher-order Programming

Exercise 3

For the following programs, find out whether they will exhause time or space resources (or both):

1) Will it exhaust time/space resources or both?

```
function loop(x) {
    return loop(x);
}
loop(0);
```

Higher-order Programming

Exercise 3

For the following programs, find out whether they will exhause time or space resources (or both):

2) Will it exhaust time/space resources or both?

```
function loop2(x) {
    return loop2(loop2(x));
}
loop2(0);
```

Higher-order Programming

Exercise 3

For the following programs, find out whether they will exhause time or space resources (or both):

3) Will it exhaust time/space resources or both?

```
function recur(x) {
    return x(x);
}
recur(x => x(x(x)));
```

Studio Group Problems

Let's discuss them now.

Overview

1. More about recursion
   - From last week
   - Wishful thinking
   - Examples
2. Higher-order programming
   - Before we start
   - To understand higher-order programming
   - To use higher-order programming
   - Exercises
3. Language processing
   - Family of programming languages
   - From low-level to high-level
   - Compilation & interpretation

Language Processing

What does a programming language do?

- A programming language is a formal language that specifies a set of instructions that can be used to produce various kinds of output.
- Programming languages consist of instructions for a computer.
- Programming languages are used to create programs that implement specific algorithms.

Language Processing

History of programming languages

- *1940s*: ENIAC coding system
- *1950s*: Fortran, Lisp, Algol 58
- *1960s*: CPL, BASIC
- *1970s*: C, Pascal, Smalltalk, Prolog, Scheme, SQL
- *1980s*: C++, Erlang, Perl
- *1990s*: Haskell, Python, VB, Ruby, Lua, Java, JavaScript, PHP
- *2000s*: C#, .NET, F#, Go, Swift
- ...

Language Processing

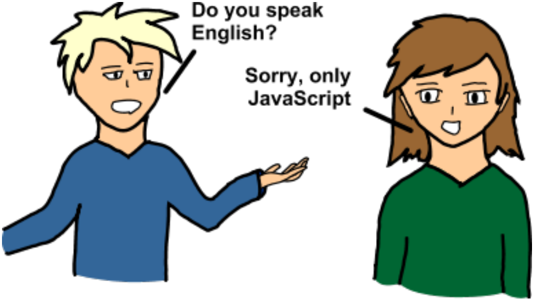
How to classify programming languages

- *According to programming paradigm*: functional, object-oriented, procedural, declarative, imperative, ...;
- *According to the way of execution*: compile, interpret;
- *According to the field of usage*: web, mobile, database, security, design, scientific calculation, ...;
- *According to typing system*: typed/untyped, static/dynamic typing, strong/weak typing, ...;
- ...

Language Processing

How does the machine understand programs?

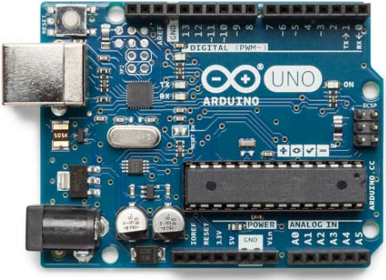
- No, computers actually does not understand the programs written by programmers.



Language Processing

What does the machine understand?

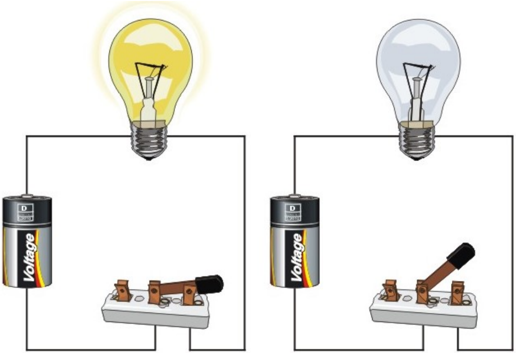
- Computers only understand byte-language (language of 0s and 1s).
- This is because computer is an electronic machine, essentially, a lot of electrical circuits.
- For each circuit, there are only 2 states: *on/off (have/no current)*.



Language Processing

What does “on/off” mean?

- They simply refer to whether the circuit has current inside, i.e., whether open circuit or not.



Language Processing

What is the work of CPU?

- Each CPU has a set of basic operations () that it can perform directly.
- The instructions that a CPU can execute are determined by its instruction set architecture (ISA).
- CPU can execute a program only if it is converted to machine code.

Language Processing

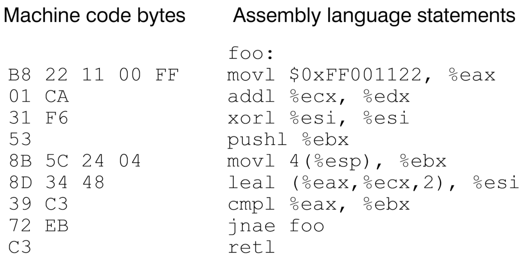
Instruction set architecture (ISA)

- There are mainly two families, following CISC and RISC paradigm.
- x86/x86-64 is widely used on desktops and personal computers.
- ARM is widely used on mobile devices, like smart phones, iPad, etc.

Language Processing

Assembly language

- Machine code is not human-readable.
- To make life easier, people invent **assembly languages** which use mnemonics (labels and symbols) to replace some 0s and 1s.
- Assembly code can be converted to *executable* machine code using a utility called *assembler*.



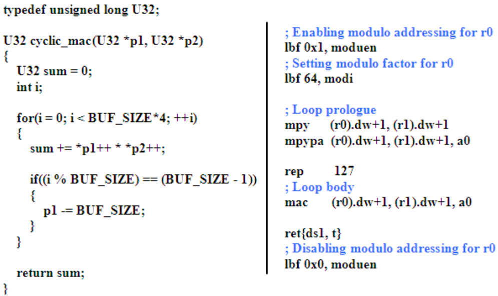
```
Machine code bytes       Assembly language statements

                         foo:
B8 22 11 00 FF           movl $0xFF001122, %eax
01 CA                    addl %ecx, %edx
31 F6                    xorl %esi, %esi
53                       pushl %ebx
8B 5C 24 04              movl 4(%esp), %ebx
8D 34 48                 leal (%eax,%ecx,2), %esi
39 C3                    cmpl %eax, %ebx
72 EB                    jnae foo
C3                       retl
```

Language Processing

High-level language

- However, as you can see, assembly code is *still* very hard to maintain.
- Therefore, people have invented more powerful languages later. They usually use some English words as syntax, like C, Java and JavaScript.
- We almost only use high-level languages nowadays.



```
typedef unsigned long U32;

U32 cyclic_mac(U32 *p1, U32 *p2)
{
   U32 sum = 0;
   int i;

   for(i = 0; i < BUF_SIZE*4; ++i)
   {
      sum += *p1++ * *p2++;

      if((i % BUF_SIZE) == (BUF_SIZE - 1))
      {
         p1 -= BUF_SIZE;
      }
   }

   return sum;
}
```

```
; Enabling modulo addressing for r0
lbf 0x1, moduen
; Setting modulo factor for r0
lbf 64, modi

; Loop prologue
mpy     (r0).dw+1, (r1).dw+1
mpypa  (r0).dw+1, (r1).dw+1, a0

rep      127
; Loop body
mac     (r0).dw+1, (r1).dw+1, a0

ret{ds1, t}
; Disabling modulo addressing for r0
lbf 0x0, moduen
```

Language Processing

The "gap" now...

- For CPU: they only understand low-level machine code;
- For programmers: they only write codes in high-level languages.

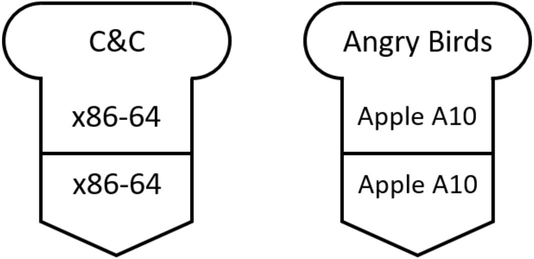
Solution

- Interpreter: a program that can execute another program written in high-level languages, like JavaScript, Python, Ruby, etc.
- Compiler: a program that compiles high-level language progams into *executable* low-level languages, and waits for it to be executed, like C, C++, etc.
- Translator: a program that translates high-level languages to other languages, like TypeScript, etc.

Language Processing

T-diagrams - direct executable

- You can directly write programs in machine code and they will be able to execute directly (although your life will be painful).



C&C

x86-64

x86-64

Angry Birds

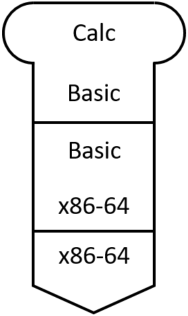
Apple A10

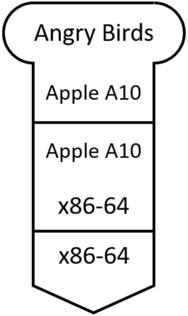
Apple A10

Language Processing

Use T-diagrams - interpreter

- However, in most cases, you should write programs in high-level languages and use an interpreter to execute them.



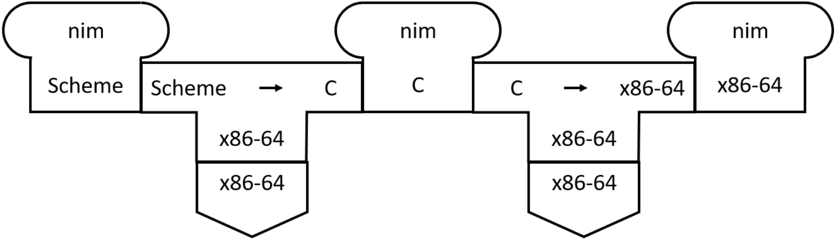


(Hardware emulation)

Language Processing

Use T-diagrams - compiler

- For some other languages, they need a compiler to compile them to low-level languages to be able to execute.
- The translation may be done in multiple steps.



(Two-stage compilation)

Language Processing

Cross platform

- Machine code may be different for different CPUs (x86/64, ARM).
- That means, the same program cannot be used across different paltforms (devices running on different hardware).
- Is it possible for the same program to run anywhere?

Language Processing

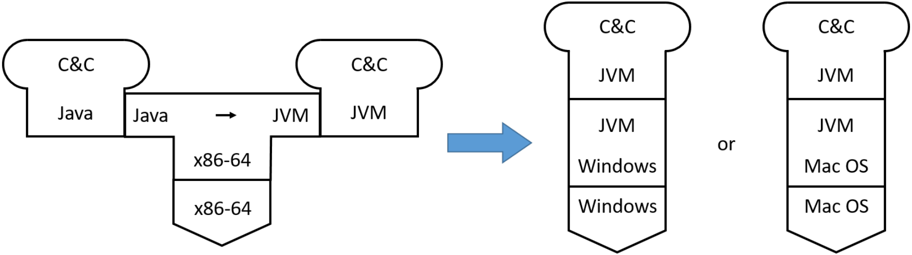
Solution - virtual machine (VM)

- We implement the same virtual machine (VM) for all platforms.
- Therefore, other programs will be able to run anywhere as long as they are converted into the "machine code" of this VM.

Language Processing

Use T-diagrams - VM

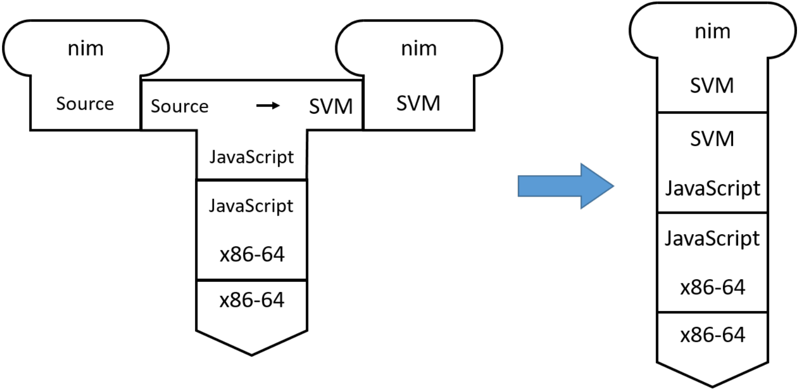
- A very famous example: Java Virtual Machine (JVM)



Language Processing

Use T-diagrams - VM

- Not that famous example: “old” Source Virtual Machine (SVM)



Language Processing

Recommended modules at SoC

- CS2104 Programming Language Concepts
- CS4212 Compiler Design
- CS6202 Advanced Topics in Programming Languages

Caution

- Conceptual-oriented;
- Abstract and theoretical.

End

The End

Copyright