L12C: Register Machines

CS1101S: Programming Methodology

Boyd Anderson

November 3, 2021

Readings

- Textbook Chap. 5

Outline

Register Machines

Demo and Examples

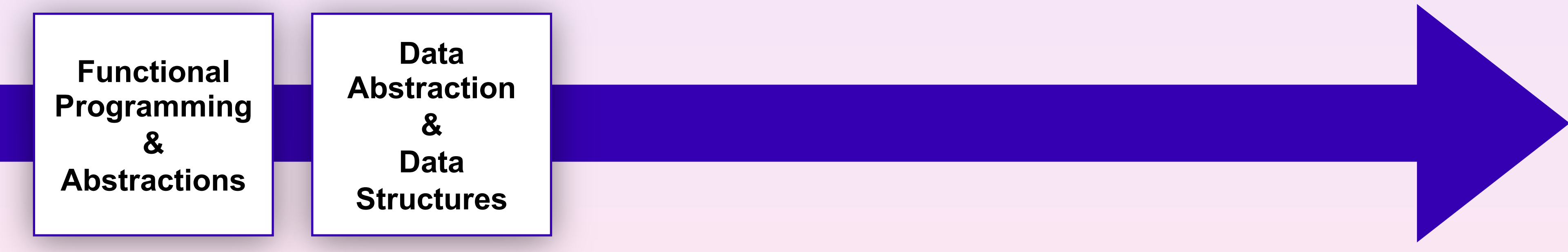
Storage Allocation and Garbage Collection

Our Journey So Far

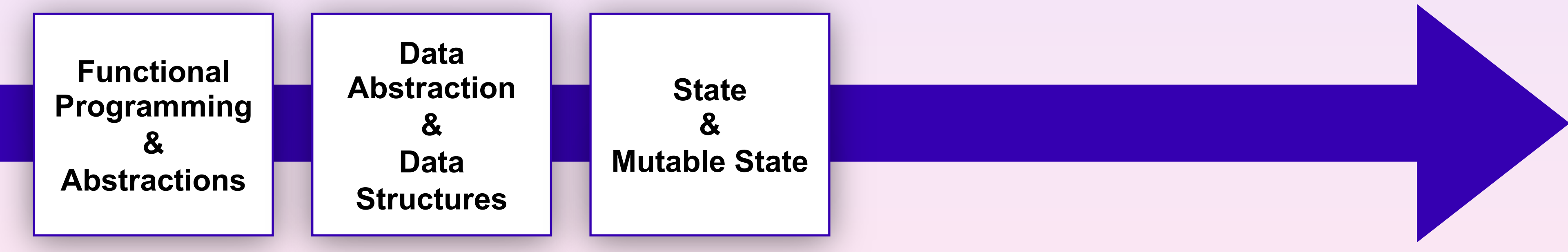
Our Journey So Far

Functional Programming & Abstractions

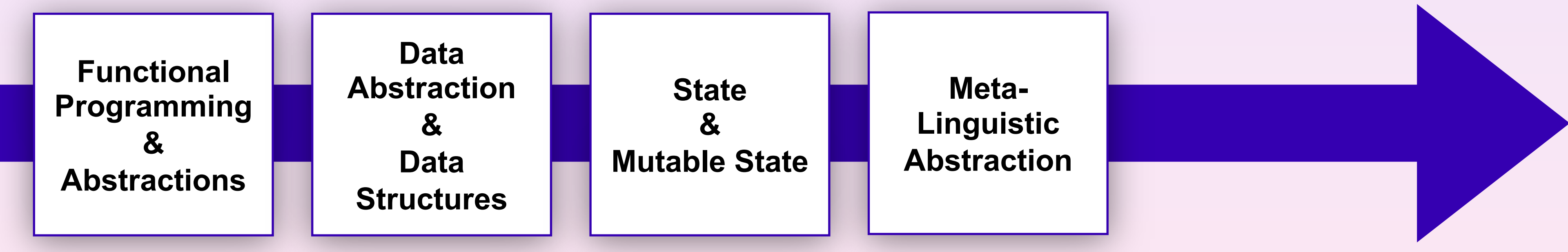
Our Journey So Far



Our Journey So Far



Our Journey So Far



Our Journey So Far

Functional Programming & Abstractions

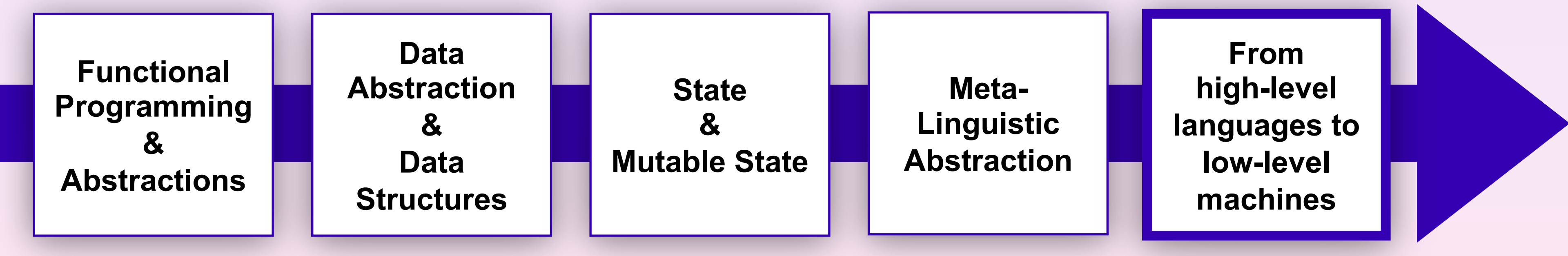
Data Abstraction & Data Structures

State & Mutable State

Meta-Linguistic Abstraction

From high-level languages to low-level machines

Our Journey So Far



Can we use high-level programming to explain low-level machines?

Abstract Machines

At some point your programs **have** to run on some hardware.

(no matter how many meta-meta-meta…circular evaluators you implement)

To understand how your program runs on hardware we need to come up with a model.

An **abstract machine** is a theoretical model of computer hardware. We will look at one particular type of abstract machine…

Register Machines

Register Machine: An idealised computing machine consisting of a fixed set of storage *registers* and set of *instructions* for operating on them.

The register machine sequentially executes *instructions*.

A typical register machine instruction applies a primitive operation to the contents of some registers and assigns the result to another register.

Example: Greatest Common Divisor

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Example: Greatest Common Divisor

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Example: Greatest Common Divisor

Data

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Example: Greatest Common Divisor

Data

Branching

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Example: Greatest Common Divisor

Data   Branching   Modulo operator

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Example: Greatest Common Divisor

Data

Branching

Modulo operator

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Recursion

Example: Greatest Common Divisor

Data Branching Modulo operator

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Comparison operator Recursion

Example: Greatest Common Divisor

Data

Branching

Modulo operator

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Return value

Comparison operator

Recursion

A Hypothetical Machine Language

A Hypothetical Machine Language

Program: A sequence of instructions and labels.

A Hypothetical Machine Language

Program: A sequence of instructions and labels.

Labels: A named place in the sequence to which it is possible to "jump to".

A Hypothetical Machine Language

Program: A sequence of instructions and labels.

Labels: A named place in the sequence to which it is possible to "jump to".

Registers: Holders of values that can be *read* and *updated*.

A Hypothetical Machine Language

Program: A sequence of instructions and labels.

Labels: A named place in the sequence to which it is possible to "jump to".

Registers: Holders of values that can be *read* and *updated*.

Instructions:

- **Test** — check a boolean condition and remember result
- **Branch** — jump to a label if a test succeeded
- **Go to** — jump to a label unconditionally
- **Assign** — update the value of a register

A Hypothetical Machine Language

Program: A sequence of instructions and labels.

Labels: A named place in the sequence to which it is possible to "jump to".

Registers: Holders of values that can be *read* and *updated*.

Instructions:

- **Test** — check a boolean condition and remember result
- **Branch** — jump to a label if a test succeeded
- **Go to** — jump to a label unconditionally
- **Assign** — update the value of a register

Primitive operations: =, +, –, print, take remainder, …

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

```
begin:
    check if b === 0
    if so go to done, otherwise…
    assign t to the value of a % b
    assign a to the value of b
    assign b to the value of t
    go to begin
done:
```

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

```
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
```

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Labels

```
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
```

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Labels

```
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
```

Registers

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Labels

```
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
```

Instructions

Registers

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

Primitive Operations

Labels

```
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
```

Instructions

Registers

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}
```

```
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
```

Loops in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, a % b);
}

while (true) {
    display(gdc(prompt_n(), prompt_n()));
}
```

Assume that `prompt_n()` prompts the user for a number.

Now we have an infinite loop? How can we handle that?

Loops in this Hypothetical Machine Language

```
gcd_loop:
    assign(a, op(read))
    assign(b, op(read))
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
    perform(op(print), reg(a))
    go_to(label(gcd_loop))
```

Loops in this Hypothetical Machine Language

```
gcd_loop:
    assign(a, op(read))
    assign(b, op(read))
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
    perform(op(print), reg(a))
    go_to(label(gcd_loop))
```

Read into register

Loops in this Hypothetical Machine Language

```
gcd_loop:
    assign(a, op(read))
    assign(b, op(read))
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
    perform(op(print), reg(a))
    go_to(label(gcd_loop))
```

GCD in this Hypothetical Machine Language

```
gcd_loop:
    assign(a, op(read))
    assign(b, op(read))
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
    perform(op(print), reg(a))
    go_to(label(gcd_loop))
```

GCD in this Hypothetical Machine Language

```
gcd_loop:
    assign(a, op(read))
    assign(b, op(read))
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
    perform(op(print), reg(a))
    go_to(label(gcd_loop))
```

Modulo Operation

GCD in this Hypothetical Machine Language

```
gcd_loop:
    assign(a, op(read))
    assign(b, op(read))
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
    perform(op(print), reg(a))
    go_to(label(gcd_loop))
```

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, rem(a,b));
}

function rem(a, b) {
    return a < b ? a : rem(a - b, b);
}
```

Show in Playground

```
test_rem:
    test(op(<), reg(a), reg(b))
    branch(label(rem_done))
    assign(a, op(-), reg(a), reg(b))
    go_to(label(test_rem))
rem_done:
    // result in reg(a)
```

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, rem(a,b));
}

function rem(a, b) {
    return a < b ? a : rem(a - b, b);
}

while (true) {
    display(gdc(prompt_n(), prompt_n());
}
```

```
gcd_loop:
    assign(a, op(read))
    assign(b, op(read))
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    go_to(label(test_rem))
rem_done:
    assign(t, reg(a))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
    perform(op(print), reg(a))
    go_to(label(gcd_loop))
```

```
test_rem:
    test(op(<), reg(a), reg(b))
    branch(label(rem_done))
    assign(a, op(-), reg(a), reg(b))
    go_to(label(test_rem))
```

GCD in this Hypothetical Machine Language

```
function gcd(a, b) {
    return b === 0 ? a : gcd(b, rem(a,b));
}

function rem(a, b) {
    return a < b ? a : rem(a - b, b);
}

while (true) {
    display(gdc(prompt_n(), prompt_n());
}
```

But what if we wanted to use rem somewhere else?

```
gcd_loop:
    assign(a, op(read))
    assign(b, op(read))
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    go_to(label(test_rem))
rem_done:
    assign(t, reg(a))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
    perform(op(print), reg(a))
    go_to(label(gcd_loop))
```

```
test_rem:
    test(op(<), reg(a), reg(b))
    branch(label(rem_done))
    assign(a, op(-), reg(a), reg(b))
    go_to(label(test_rem))
```

The Need for Subroutines

```
test_rem:
    test(op(<), reg(a), reg(b))
    branch(label(rem_done))
    assign(a, op(-), reg(a), reg(b))
    go_to(label(test_rem))
rem_done:
    // result in reg(a)
```

The Need for Subroutines

How do we know where to return?

Can we be sure that registers a and b hold the right values?

```
test_rem:
    test(op(<), reg(a), reg(b))
    branch(label(rem_done))
    assign(a, op(-), reg(a), reg(b))
    go_to(label(test_rem))
rem_done:
    // result in reg(a)
```

```
test_rem2:
    test(op(<), reg(c), reg(d))
    branch(label(rem_done2))
    assign(c, op(-), reg(c), reg(d))
    go_to(label(test_rem2))
rem_done2:
    // result in reg(c)
```

Do we need to duplicate *rem*?

A New Idea:

Why not allow labels to be part of program data

Add a continue register — from where a computation should continue

```
test_rem:
    test(op(<), reg(a), reg(b))
    branch(reg(continue))
    assign(a, op(-), reg(a), reg(b))
    go_to(label(test_rem))
    // result in reg(a)
```

“Calling” A Subroutine

```
gcd1_loop:
    assign(a, op(read))
    assign(b, op(read))
test_gcd1:
    test(op(=), reg(b), constant(0))
    branch(label(gcd1_done))
    assign(continue, label(gcd_cont))
    go_to(label(test_rem))
gcd_cont:
    assign(t, reg(a))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd1))
gcd1_done:
    perform(op(print), reg(a))
    go_to(label(gcd1_loop))
```

```
test_rem:
    test(op(<), reg(a), reg(b))
    branch(reg(continue))
    assign(a, op(-), reg(a), reg(b))
    go_to(label(test_rem))
    // result in reg(a)
```

What about deferred operations?

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n - 1);
}
```

What about deferred operations?

Deferred Operation

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n - 1);
}
```

What about deferred operations?

Deferred Operation

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n - 1);
}
```

```
test_fac:
    test(op(=), reg(a), constant(1))
    branch(label(fac_done))
    assign(b, reg(a))
    assign(a, op(-), reg(a), constant(1))
    go_to(label(test_fac))
fac_done:
    assign(a, op(*), reg(a), reg(b))
    perform(op(print), reg(a))
```

What about deferred operations?

Deferred Operation

```
function factorial(n) {
    return n === 1 ? 1 : n * factorial(n - 1);
}
```

```
test_fac:
    test(op(=), reg(a), constant(1))
    branch(label(fac_done))
    assign(b, reg(a))
    assign(a, op(-), reg(a), constant(1))
    go_to(label(test_fac))
fac_done:
    assign(a, op(*), reg(a), reg(b))
    perform(op(print), reg(a))
```

How do we keep track of an unknown number of deferred operations?

Supporting Deferred Operations with a Stack

Stack data structure

Each *frame* in the stack can store the current value of one or more registers.

When calling a function: **save** necessary register values and push a new frame

When returning from a function: **restore** necessary register values for continuing

Allows us to reuse the same factorial machine code for every factorial subproblem!

New register: val

Used to hold return values from function calls.

Stack Example

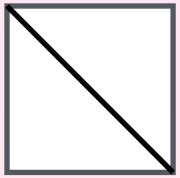
```
function make_stack() {
    function push(x) {
        stack = pair(x, stack);
        return "done";
    }

    function pop() {
        if (is_null(stack)) {
            error("Empty stack: POP");
        } else {
            const top = head(stack);
            stack = tail(stack);
            return top;
        }
    }
    ... //I have shortened this for this slide.
}
```

[Show in Playground]

Stack Example

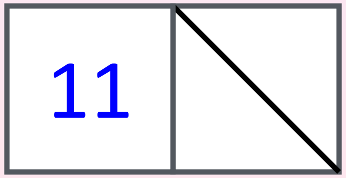
Stack Example

```
let stack = make_stack();
```



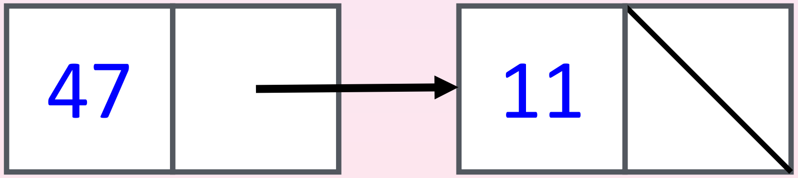
Stack Example

```
let stack = make_stack();
stack(“push”)(11);
```



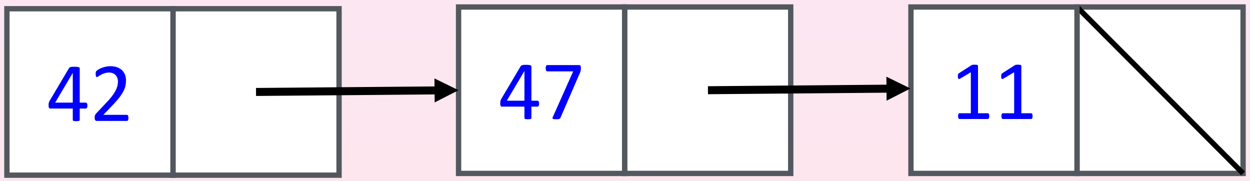
Stack Example

```
let stack = make_stack();
stack(“push”)(11);
stack(“push”)(47);
```



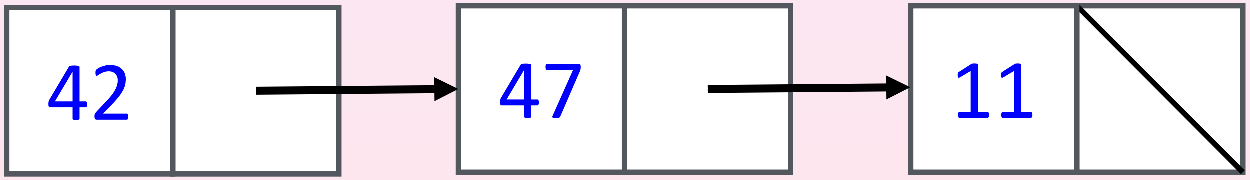
Stack Example

```
let stack = make_stack();
stack("push")(11);
stack("push")(47);
stack("push")(42);
```



Stack Example

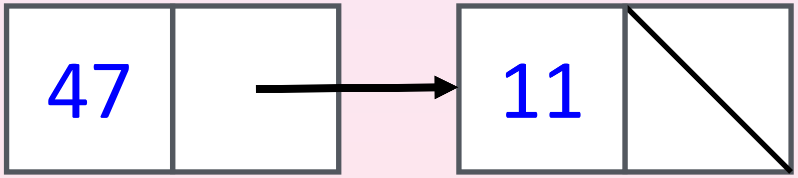
```
let stack = make_stack();
stack("push")(11);
stack("push")(47);
stack("push")(42);
```



Stack Example

```
let stack = make_stack();
stack(“push”)(11);
stack(“push”)(47);
stack(“push”)(42);
stack(“pop”);
```

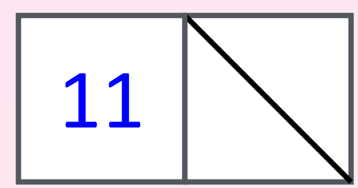
42



Stack Example

```
let stack = make_stack();
stack(“push”)(11);
stack(“push”)(47);
stack(“push”)(42);
stack(“pop”);
stack(“pop”);
```

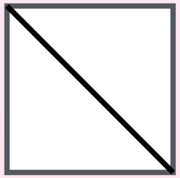
47



Stack Example

```
let stack = make_stack();
stack("push")(11);
stack("push")(47);
stack("push")(42);
stack("pop");
stack("pop");
stack("pop");
```

11



Supporting Deferred Operations with a Stack

When calling a function: **save** necessary register values and push a new frame

When returning from a function: **restore** necessary register values for continuing

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue | L | **a** | 5 | **val** | |

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue B **a** 4 **val**

The Stack

a 5 continue L

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue B **a** 3 **val**

The Stack

a	5	continue	L
a	4	continue	B

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue B **a** 2 **val**

The Stack

a	5	continue	L
a	4	continue	B
a	3	continue	B

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue B **a** 1 **val**

The Stack

a	5	continue	L
a	4	continue	B
a	3	continue	B
a	2	continue	B

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue B **a** 1 **val** 1

The Stack

a	5	continue	L
a	4	continue	B
a	3	continue	B
a	2	continue	B

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue	a	val
B	2	2

The Stack

a	continue
5	L
4	B
3	B

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue B **a** 3 **val** 6

The Stack

a	5	continue	L
a	4	continue	B

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue B **a** 4 **val** 24

The Stack

a	5	continue	L

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Factorial Example

continue L **a** 5 **val** 120

The Stack

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Simulating a Register Machine in Source

SICP provides a register machine simulator in Chapter 5

Create the register machine (specify registers, primitives, and instructions). Then:

Set the registers with the values you want

Start the machine

Get the register value(s) with the result

A Simple Example

```
function simple_machine() {
   return make_machine(
      list("a", "b"),
      list(list("+", binary_function((a, b) => a + b))),
      list(assign("a", list(op("+"), reg("a"), reg("b")))));
}

const m = simple_machine();

display(set_register_contents(m, "a", 206));
display(set_register_contents(m, "b", 40));
display(start(m));
display(get_register_contents(m, "a"));
```

Show in Playground

A Simple Example

```
function simple_machine() {
   return make_machine(
      list("a", "b"),
      list(list("+", binary_function((a, b) => a + b))),
      list(assign("a", list(op("+"), reg("a"), reg("b")))));
}

const m = simple_machine();

display(set_register_contents(m, "a", 206));
display(set_register_contents(m, "b", 40));
display(start(m));
display(get_register_contents(m, "a"));
```

A list of registers

[Show in Playground]

A Simple Example

```
function simple_machine() {
   return make_machine(
      list("a", "b"),
      list(list("+", binary_function((a, b) => a + b))),
      list(assign("a", list(op("+"), reg("a"), reg("b")))));
}

const m = simple_machine();

display(set_register_contents(m, "a", 206));
display(set_register_contents(m, "b", 40));
display(start(m));
display(get_register_contents(m, "a"));
```

A list of registers

A list of primitive operations

Show in Playground

A Simple Example

```
function simple_machine() {
   return make_machine(
      list("a", "b"),
      list(list("+", binary_function((a, b) => a + b))),
      list(assign("a", list(op("+"), reg("a"), reg("b")))));
}

const m = simple_machine();

display(set_register_contents(m, "a", 206));
display(set_register_contents(m, "b", 40));
display(start(m));
display(get_register_contents(m, "a"));
```

A list of registers

A list of primitive operations

A list of instructions

[Show in Playground]

A Simple Example

```
function simple_machine() {
   return make_machine(
      list("a", "b"),
      list(list("+", binary_function((a, b) => a + b))),
      list(assign("a", list(op("+"), reg("a"), reg("b")))));
}

const m = simple_machine();

display(set_register_contents(m, "a", 206));
display(set_register_contents(m, "b", 40));
display(start(m));
display(get_register_contents(m, "a"));
```

A list of registers

A list of primitive operations

A list of instructions

Create Machine

[Show in Playground]

A Simple Example

```
function simple_machine() {
    return make_machine(
        list("a", "b"),
        list(list("+", binary_function((a, b) => a + b))),
        list(assign("a", list(op("+"), reg("a"), reg("b")))));
}

const m = simple_machine();

display(set_register_contents(m, "a", 206));
display(set_register_contents(m, "b", 40));
display(start(m));
display(get_register_contents(m, "a"));
```

A list of registers

A list of primitive operations

A list of instructions

Create Machine

Set registers

[Show in Playground]()

A Simple Example

```
function simple_machine() {
   return make_machine(
      list("a", "b"),
      list(list("+", binary_function((a, b) => a + b))),
      list(assign("a", list(op("+"), reg("a"), reg("b")))));
}

const m = simple_machine();

display(set_register_contents(m, "a", 206));
display(set_register_contents(m, "b", 40));
display(start(m));
display(get_register_contents(m, "a"));
```

A list of registers

A list of primitive operations

A list of instructions

Create Machine

Set registers

A list of instructions

[Show in Playground]()

A Simple Example

```
function simple_machine() {
   return make_machine(
      list("a", "b"),
      list(list("+", binary_function((a, b) => a + b))),
      list(assign("a", list(op("+"), reg("a"), reg("b")))));
}

const m = simple_machine();

display(set_register_contents(m, "a", 206));
display(set_register_contents(m, "b", 40));
display(start(m));
display(get_register_contents(m, "a"));
```

A list of registers

A list of primitive operations

A list of instructions

Create Machine

Set registers

A list of instructions

Get register/result

[Show in Playground]()

Inside the make_machine function

```
function make_machine(register_names, ops, controller_text) {
    const machine = make_new_machine();
    map(reg_name => machine("allocate_register")(reg_name),
           register_names);

    machine("install_operations")(ops);
    machine("set_instructions")(assemble(controller_text, machine));
    return machine;
}
```

Inside the make_machine function

```
function make_machine(register_names, ops, controller_text) {
    const machine = make_new_machine();
    map(reg_name => machine("allocate_register")(reg_name),
           register_names);

    machine("install_operations")(ops);
    machine("set_instructions")(assemble(controller_text, machine));
    return machine;
}
```

Create all the registers

Inside the make_machine function

```
function make_machine(register_names, ops, controller_text) {
    const machine = make_new_machine();
    map(reg_name => machine("allocate_register")(reg_name),
           register_names);

    machine("install_operations")(ops);
    machine("set_instructions")(assemble(controller_text, machine));
    return machine;
}
```

Create all the registers

Add primitives

Inside the make_machine function

```
function make_machine(register_names, ops, controller_text) {
    const machine = make_new_machine();
    map(reg_name => machine("allocate_register")(reg_name),
           register_names);

    machine("install_operations")(ops);
    machine("set_instructions")(assemble(controller_text, machine));
    return machine;
}
```

Create all the registers

Add primitives

Convert controller instructions into machine code. Extract labels, bind pc/flag registers, etc

Inside the make_new_machine function

```
function make_new_machine() {
    const pc = make_register("pc");
    const flag = make_register("flag");
    const stack = make_stack();
    let instructions = null;
    let the_ops = list(list("initialize_stack",
                            () => stack("initialize")));
    let register_table = list(list("pc", pc), list("flag", flag));
    ...
    return dispatch;
}
```

Built-in Registers

pc — what to do next

flag — holds test results

And a table to keep track of them

Initialise Stack

To save register values as before

Instruction Sequence

Empty to begin with

Inside the make_register function

```
function make_register(name) {
    let contents = "*unassigned*";

    function dispatch(message) {
        if (message === "get") {
            return contents;
        } else {
            if (message === "set") {
                return value => { contents = value; };
            } else {
                error(message, "Unknown request: REGISTER");
            }
        }
    }
    return dispatch;
}
```

Inside the make_register function

```
function get_contents(register) {
    return register("get");
}

function set_contents(register, value) {
    return register("set")(value);
}
```

Four functions in make_new_machine

```
function make_new_machine() {
    ...
    function allocate_register(name) { … }
    function lookup_register(name) { … }
    function execute() { … }
    function dispatch(message) { … }
    ...
}
```

allocate_register

Creates a new register and remembers the names to ensure registers have unique names

lookup_register

Look up the value of a register and return it

Inside the execute function

```
function execute() {
    const insts = get_contents(pc);

    if (is_null(insts)) {
        return "done";
    } else {
        const proc =
            instruction_execution_proc(head(insts));
        proc();
        return execute();
    }
}
```

Inside the dispatch function

```
function dispatch(message) {
    return message === "start"
        ? () => { set_contents(pc, instructions);
                  return execute(); }
        : message === ”set_instructions"
        ? seq => { instructions = seq; }
        : message === "allocate_register"
        ? allocate_register
        : message === "get_register"
        ? lookup_register
        : message === "install_operations"
        ? ops => { the_ops = append(the_ops, ops); }
        : message === "stack"
        ? stack
        : message === "operations"
        ? the_ops
        : error(message, "Unknown request");
}
```

Starting the register machine

```
function dispatch(message) {
    return message === "start"
        ? () => { set_contents(pc, instructions);
                  return execute(); }
    ...
}

function start(machine) {
    return machine("start")();
}

start(m);
```

GCD Example in Source

```
test_gcd:
    test(op(=), reg(b), constant(0))
    branch(label(gcd_done))
    assign(t, op(%), reg(a), reg(b))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd))
gcd_done:
```

Registers: a b t

Primitive operations: = % (rem)

GCD Example in Source

```
function gcd_machine() {
   return make_machine(
      list("a", "b", "t"),
      list(list("rem", binary_function((a, b) => a % b)),
           list("=", binary_function((a, b) => a === b))),
      list("test-b",
           test(op("="), reg("b"), constant(0)),
           branch(label("gcd-done")),
           assign("t", list(op("rem"), reg("a"), reg("b"))),
           assign("a", list(reg("b"))),
           assign("b", list(reg("t"))),
           go_to(label("test-b")),
           "gcd-done"));
}
```

Show in Playground

GCD with REM Subroutine in Source

```
test_gcd1:
    test(op(=), reg(b), constant(0))
    branch(label(gcd1_done))
    assign(continue, label(gcd_cont))
    go_to(label(test_rem))
gcd_cont:
    assign(t, reg(a))
    assign(a, reg(b))
    assign(b, reg(t))
    go_to(label(test_gcd1))
test_rem:
    test(op(<), reg(a), reg(b))
    branch(reg(continue))
    assign(a, op(-), reg(a), reg(b))
    go_to(label(test_rem))
gcd1_done:
    // result in reg(a)
```

Registers: a b t

Primitive operations:
= * - <

GCD with REM Subroutine in Source

```
function gcd_machine() {
   return make_machine(
      list("a", "b", "t"),
      list(list("=", binary_function((a, b) => a === b)),
      list("<", binary_function((a, b) => a < b)),
      list("-", binary_function((a, b) => a - b))),
      list("test-b",
           test(op("="), reg("b"), constant(0)),
           branch(label("gcd-done")),
           go_to(label("test_rem")),
           "rem_done",
           assign("t", list(reg("a"))),
           assign("a", list(reg("b"))),
           assign("b", list(reg("t"))),
           go_to(label("test-b")),
           "test_rem",
           test(op("<"), reg("a"), reg("b")),
           branch(label("rem_done")),
           assign("a",list(op("-"),reg("a"),reg("b"))),
           go_to(label("test_rem")),
           "gcd-done"));
}
```

Show in Playground

Factorial Example

```
fac_loop:
    test(op(=), reg(a), constant(1))
    branch(label(base-case))
    save(continue)
    save(a)
    assign(a, op(-), reg(a), constant(1))
    assign(continue, label(fac_done))
    go_to(label(fac_loop))
fac_done:
    restore(a)
    restore(continue)
    assign(val, op(*), reg(a), reg(val))
    go_to(reg(continue))
base_case:
    assign(val, constant(1))
    go_to(reg(continue))
```

Registers: a, val, continue

Primitive operations:
= * -

Factorial Example

```
function fac_machine() {
    return make_machine(
       ...
       list(assign("continue",list(label("done"))),
            "fac_loop",
            test(op("="), reg("a"), constant(1)),
            branch(label("base_case")),
            save("continue"),
            save("a"),
            assign("a", list(op("-"), reg("a"), constant(1))),
            assign("continue", list(label("fac_done"))),
            go_to(label("fac_loop")),
            "fac_done",
            restore("a"),
            restore("continue"),
            assign("val", list(op("*"), reg("a"), reg("val"))),
            go_to(reg("continue")),
            "base_case",
            assign("val",list(constant(1))),
            go_to(reg("continue")),
            "done"));
}
```

[Show in Playground]()

Register Machines Summary

Using a small number of special registers…

- **val** — holds result of function calls
- **continue** — where to return after a function call (label)
- **stack** — the location of the stack where we save and restore register values
- **pc** — the ”program counter” which is an index in the sequence of instructions
- **flag** — stores the result of last test

And a small number of basic instructions…

- **assign** — update value of register
- **test** — boolean test (typically = and <)
- **branch** — jump to label if flag is set
- **go_to** — jump to a label
- **save** — save a register to the stack
- **restore** — restore a register from the stack

And a small number of basic operations.

Basic arithmetic, Less than, greater than, Sameness

We can do complex calculations! Register machines are simple but powerful.

Memory

We have discussed memory consumption in relation to orders of growth / space complexity.

Deferred operations for recursive processes. Iterative processes lets us reuse the same space

The pair function requires space to hold the head and tail of the newly created pair

Memoization, trees. let us do a memory vs. time trade-off.

Computer Memory

A computer typically has three types of memory:

Registers: very small, very fast memory inside processors

RAM: quite small, quite fast memory shared by all running programs to hold stacks, instructions and data (structures)

Disk: very large, very slow memory that holds all instructions and all data in a computer

Storing Pairs in Memory

Idea: Represent memory as two arrays, one holding all heads and one all tails

```
list(list(1, 2), 3, 4)
```

	0	1	2	3	4	5	6	7	8	9	10	…
heads												
tails												

Storing Pairs in Memory

Idea: Represent memory as two arrays, one holding all heads and one all tails

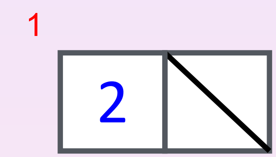
```
list(list(1, 2), 3, 4)
```

free | 1 |

	0	1	2	3	4	5	6	7	8	9	10	…
heads												
tails												

Storing Pairs in Memory

Idea: Represent memory as two arrays, one holding all heads and one all tails




```
list(list(1, 2), 3, 4)
```

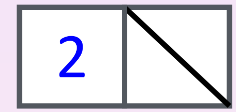
free 1

	0	1	2	3	4	5	6	7	8	9	10	...
heads												
tails												

Storing Pairs in Memory

Idea: Represent memory as two arrays, one holding all heads and one all tails

1



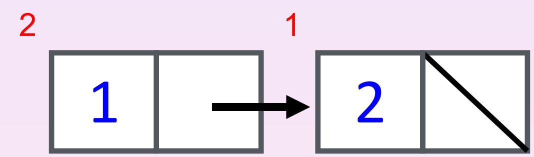

```
list(list(1, 2), 3, 4)
```

free: 1

	0	1	2	3	4	5	6	7	8	9	10	...
heads		n2										
tails		e0										

Storing Pairs in Memory

Idea: Represent memory as two arrays, one holding all heads and one all tails



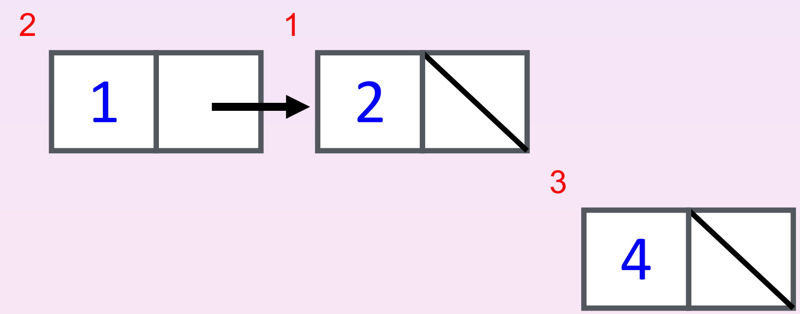

```
list(list(1, 2), 3, 4)
```

free 2

	0	1	2	3	4	5	6	7	8	9	10	...
heads		n2	n1									
tails		e0	p1									

Storing Pairs in Memory

Idea: Represent memory as two arrays, one holding all heads and one all tails



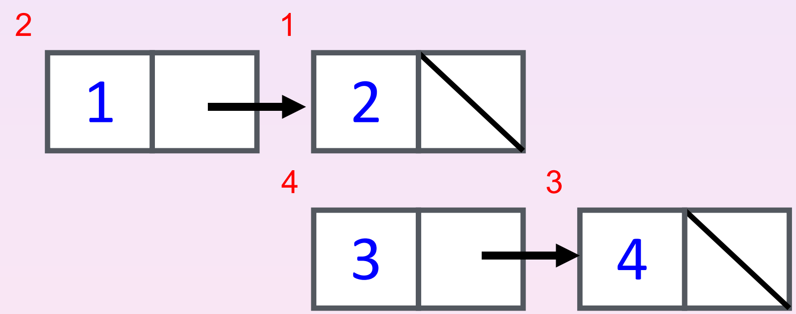

```
list(list(1, 2), 3, 4)
```

free 3

	0	1	2	3	4	5	6	7	8	9	10	…
heads		n2	n1	n4								
tails		e0	p1	e0								

Storing Pairs in Memory

Idea: Represent memory as two arrays, one holding all heads and one all tails



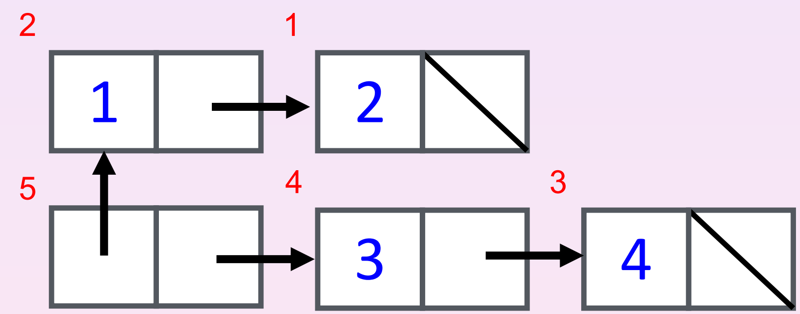

```
list(list(1, 2), 3, 4)
```

free 4

	0	1	2	3	4	5	6	7	8	9	10	...
heads		n2	n1	n4	n3							
tails		e0	p1	e0	p3							

Storing Pairs in Memory

Idea: Represent memory as two arrays, one holding all heads and one all tails




```
list(list(1, 2), 3, 4)
```

free 5

	0	1	2	3	4	5	6	7	8	9	10	...
heads		n2	n1	n4	n3	p2						
tails		e0	p1	e0	p3	p4						

Getting rid of unwanted pairs

Many programs or function calls create temporary pairs.

For example:

```
accumulate((x, y) => x + y, 0,
           filter(is_odd, enum_list(0, n)))
```

This creates two lists (enumeration list and filter list) which are not returned by the accumulator).

How can we let our computer know which pairs can be thrown away so their space can be reused?

Garbage Collection

Idea: Automatically detect “garbage objects” (pairs no longer reachable/usable from the program) and reclaim them.

Goal: To provide the illusion of “infinite” memory.

Garbage Collection is common in lots of modern programming languages.

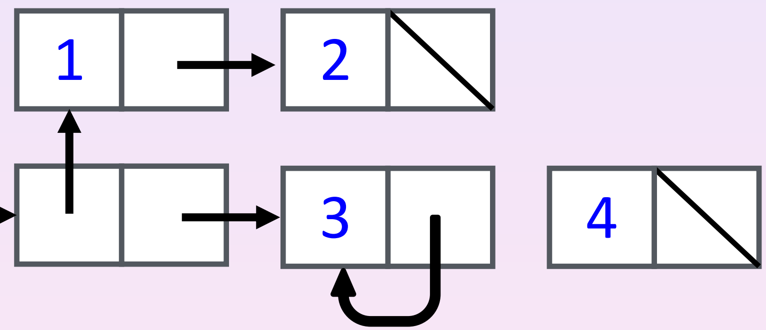
There is of course overhead and trade offs for doing this.

Trigger garbage collection when memory is full (allocation stall)

Stop-and-Copy Garbage Collector

free: 0

root: 5



Old

	0	1	2	3	4	5	6	7	8	9	10	…
heads		n2		n1	n4	p3		n3				
tails		e0		p1	e0	p7		p7				

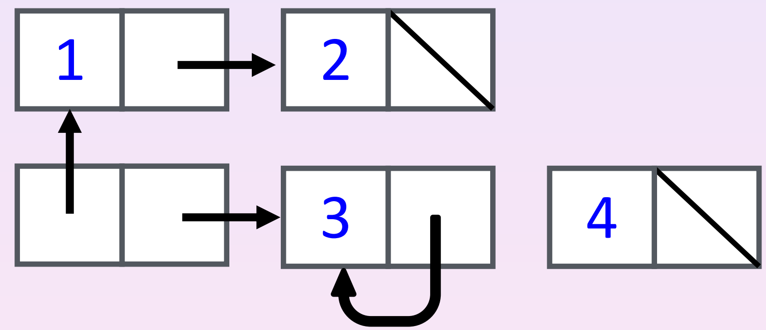
New

	0	1	2	3	4	5	6	7	8	9	10	…
new_heads												
new_tails												

Stop-and-Copy Garbage Collector

free: 1

root: 5



Old

	0	1	2	3	4	5	6	7	8	9	10	…
heads		n2		n1	n4	f0		n3				
tails		e0		p1	e0	p7		p7				

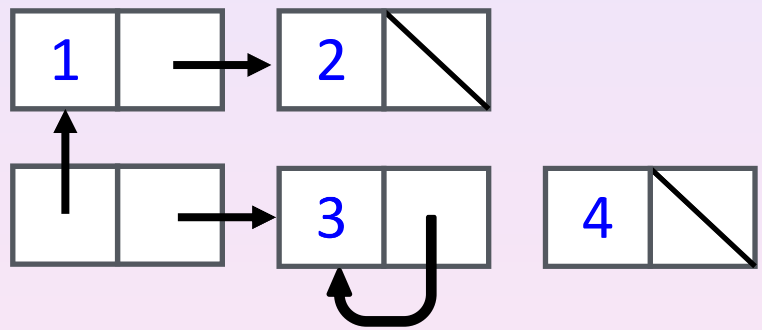
New

	0	1	2	3	4	5	6	7	8	9	10	…
new_heads												
new_tails												

Stop-and-Copy Garbage Collector

free: 2

root: 5



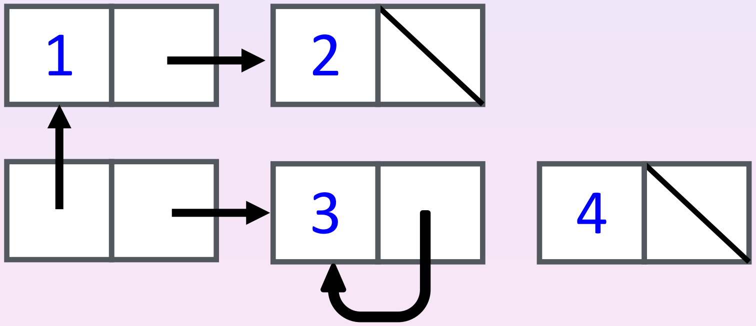
	0	1	2	3	4	5	6	7	8	9	10	...	
heads		n2		f1	n4	f0		n3					Old
tails		e0		p1	e0	p7		p7					

	0	1	2	3	4	5	6	7	8	9	10	...	
new_heads		n1											New
new_tails													

Stop-and-Copy Garbage Collector

free: 2

root: 5

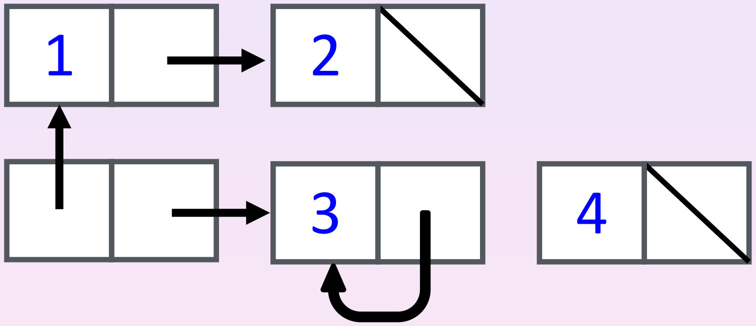


	0	1	2	3	4	5	6	7	8	9	10	...	
heads		f2		f1	n4	f0		n3					Old
tails		e0		p1	e0	p7		p7					

	0	1	2	3	4	5	6	7	8	9	10	...	
new_heads		n1	n2										New
new_tails			e0										

Stop-and-Copy Garbage Collector

free	3
root	5
val	p2



	0	1	2	3	4	5	6	7	8	9	10	...	
heads		f2		f1	n4	f0		n3					Old
tails		e0		p1	e0	p7		p7					

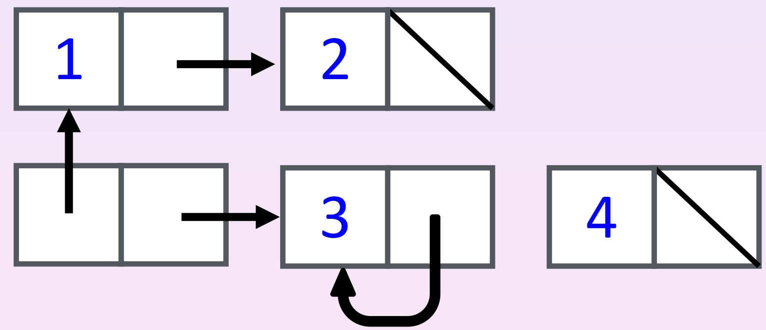
	0	1	2	3	4	5	6	7	8	9	10	...	
new_heads		n1	n2										New
new_tails		p2	e0										

Stop-and-Copy Garbage Collector

free: 3

root: 5

val: p1



Old

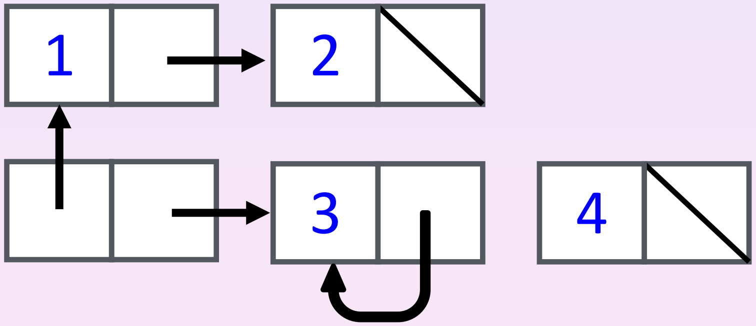
	0	1	2	3	4	5	6	7	8	9	10	...
heads		f2		f1	n4	f0		n3				
tails		e0		p1	e0	p7		p7				

New

	0	1	2	3	4	5	6	7	8	9	10	...
new_heads	p1	n1	n2									
new_tails		p2	e0									

Stop-and-Copy Garbage Collector

free	4
root	5
val	

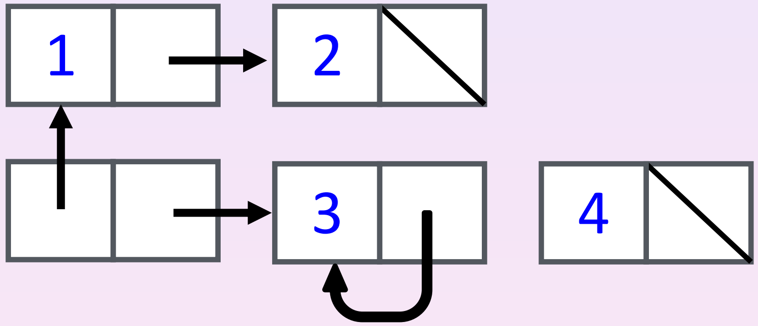


	0	1	2	3	4	5	6	7	8	9	10	...	
heads		f2		f1	n4	f0		f3					Old
tails		e0		p1	e0	p7		p7					

	0	1	2	3	4	5	6	7	8	9	10	...	
new_heads	p1	n1	n2	n3									New
new_tails		p2	e0										

Stop-and-Copy Garbage Collector

free	4
root	5
val	p3



Old

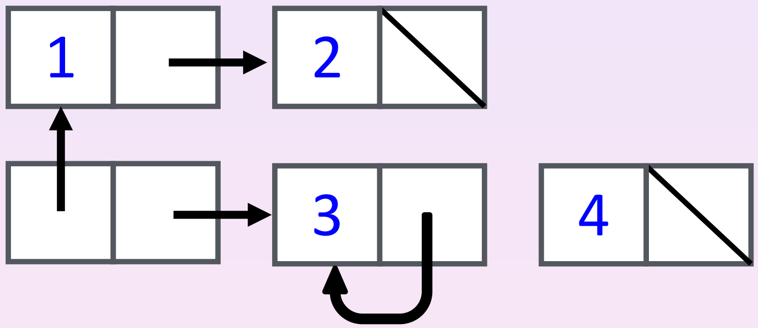
	0	1	2	3	4	5	6	7	8	9	10	...
heads		f2		f1	n4	f0		f3				
tails		e0		p1	e0	p7		p7				

New

	0	1	2	3	4	5	6	7	8	9	10	...
new_heads	p1	n1	n2	n3								
new_tails		p2	e0	p3								

Stop-and-Copy Garbage Collector

free	4
root	5
val	



Old

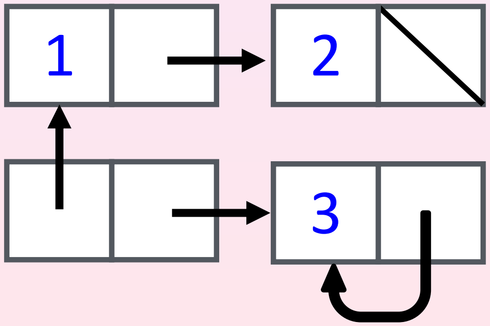
	0	1	2	3	4	5	6	7	8	9	10	…
heads		f2		f1	n4	f0		f3				
tails		e0		p1	e0	p7		p7				

New

	0	1	2	3	4	5	6	7	8	9	10	…
new_heads	p1	n1	n2	n3								
new_tails	p3	p2	e0	p3								

Stop-and-Copy Garbage Collector

	0	1	2	3	4	5	6	7	8	9	10	…
heads	p1	n1	n2	n3								
tails	p3	p2	e0	p3								



Can be implemented in just 44 instructions in the register machine simulator

Stacks and Heaps

The stack we saw for storing register values etc. is typically referred to as ”the stack” or ”the call stack”

The arrays we saw for storing pairs are typically referred to as ”the heap”

The stack is typically orders of magnitude smaller than the heap

This is why deferred operations are more costly than pairs or functions using accumulators or CPS transformation.

This is a pragmatic choice, but most programs only need a **very** small stack so it is resource-efficient

Summary

We have seen a model of a register machine explained through source.

This is a working model; you will learn more later in computer architecture modules

Only a few simple operations must be supported “out of the box” — *assign*, *test*, *branch*, *label*, *goto*…

We have seen how we can hold pairs in memory using a simple array model (and thus stacks if we implement them with pairs)

Automatic “garbage collection” is possible and strives to give the illusion of infinite memory