CSCI 334:
Principles of Programming Languages

Lecture 10: Control Structures I

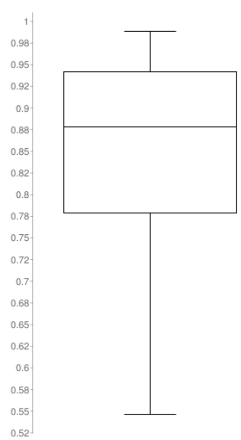
Instructor: Dan Barowy

Williams

Announcements

Graded HW2 back (except late assignments)

Announcements



HW2

Announcements

Graded HW3 back this week

Announcements

Graded HW3 back this week

Hopefully HW4 early next week

Announcements

Midterm Exam: Thurs, March 15

Mitchell Ch. 1-7

All material covered on homework

Announcements

Midterm Exam Review:

Outside of class

Probably Tuesday, March 13 at 4pm

Location TBD

Announcements

HW5 last homework before Spring Break.

Homework help session:

Thursday, March 8, 7-9 in TCL 206

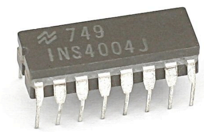
No homework help session:

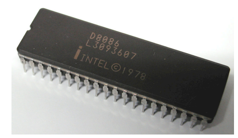
Thursday, March 15

Type Inference

Clever Uses for Types

Intel 4004 (1971)	Intel 8086 (1978)	Intel Pentium 4 (2000)

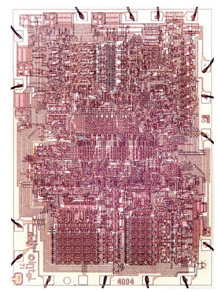


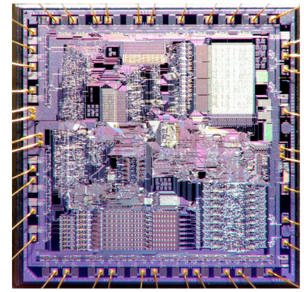


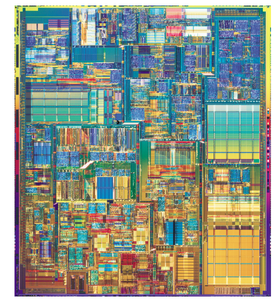
2,300 transistors

50,000 transistors

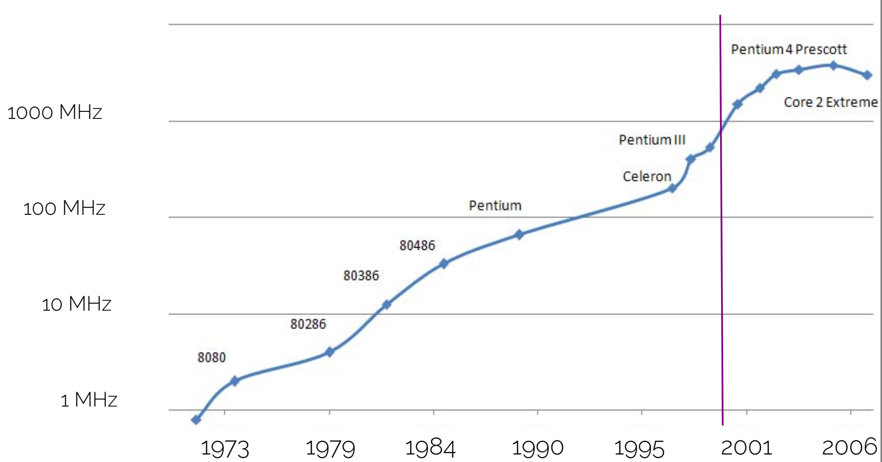
50,000,000 transistors

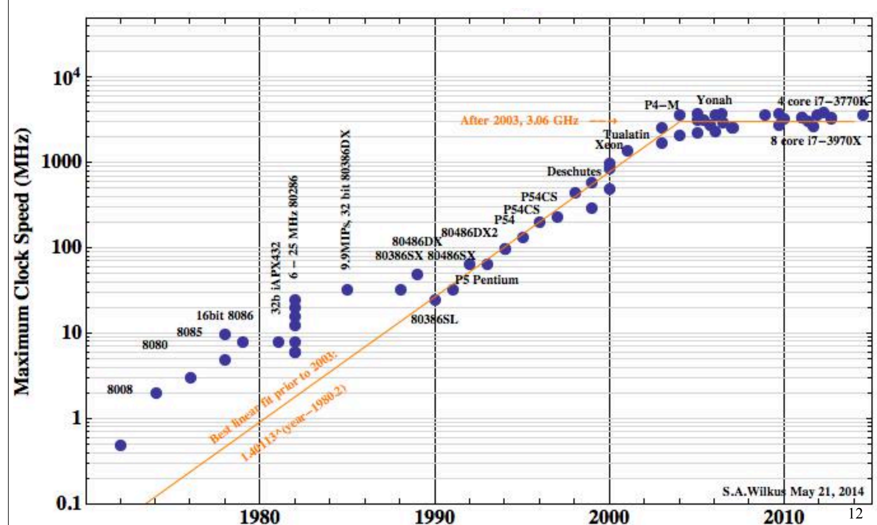




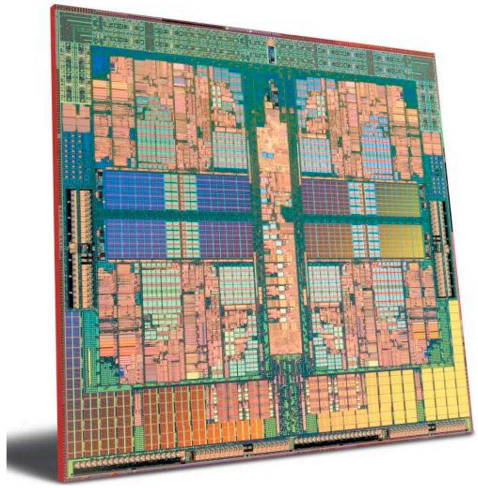


Processor Clock Speeds



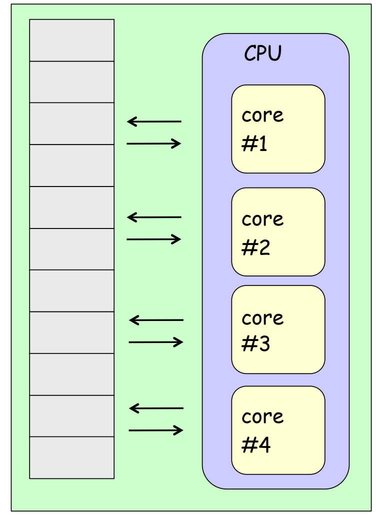


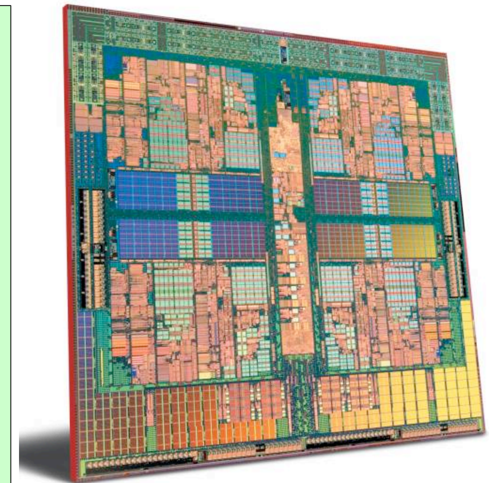
Intel Core i7 (2010)



2,000,000,000 transistors

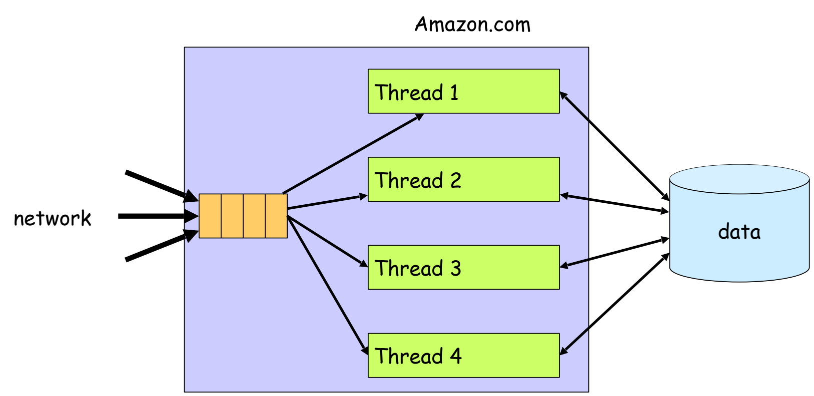
Multi-Core Chips

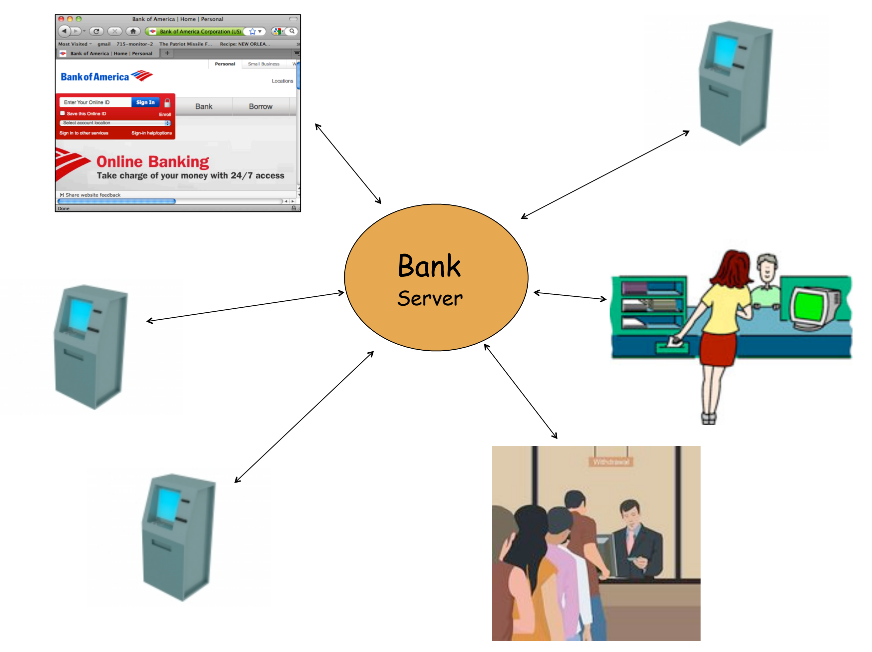




Concurrent Programming With Threads

Concurrent Programming With Threads

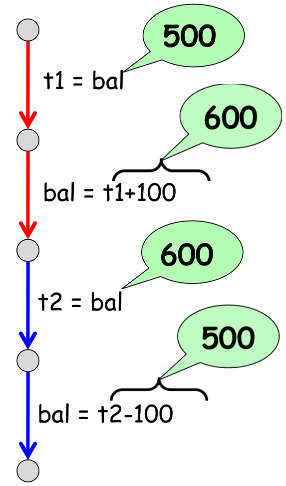




Multithreaded Program Execution

Thread A
```
...
t1 = bal;
bal = t1 + 100;
...
```

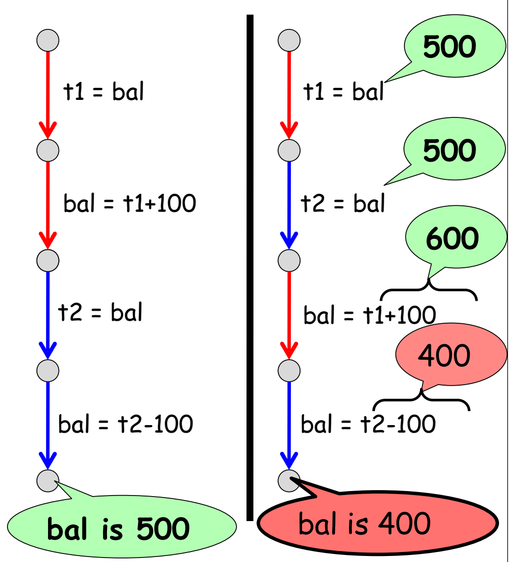
Thread B
```
...
t2 = bal;
bal = t2 - 100;
...
```



Multithreaded Program Execution

Thread A
```
...
t1 = bal;
bal = t1 + 100;
...
```

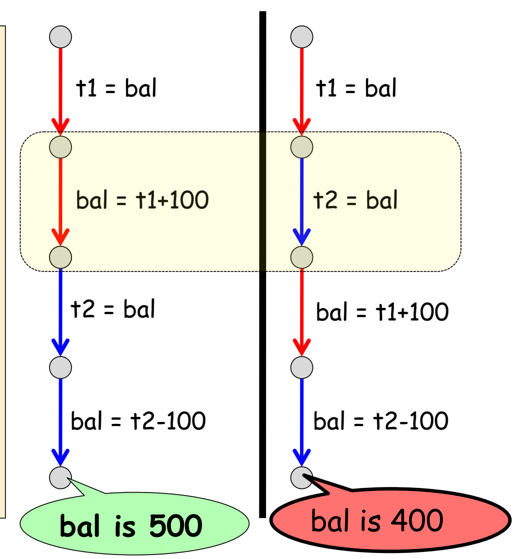
Thread B
```
...
t2 = bal;
bal = t2 - 100;
...
```



Race Condition

Thread A
```
...
t1 = bal;
bal = t1 + 100;
...
```

Thread B
```
...
t2 = bal;
bal = t2 - 100;
...
```



Race Condition

A *race condition* occurs when two or more concurrent threads:

(1) access the same variable at the same time,

(2) at least one of the threads performs a write to the variable, and

(3) the order of read/write events can cause the program to compute different results.

Avoiding Race Conditions

A *mutual exclusion barrier* (or *mutex*, or *lock*) is a concurrency control structure that prevents race conditions by limiting the possible *thread interleavings*.

Avoiding Race Conditions

```
Thread A
  acquire(m);
   t1 = bal;
   bal = t1 + 100;
  release(m);
```

```
Thread B
  acquire(m);
   t2 = bal;
   bal = t2 - 100;
  release(m);
```

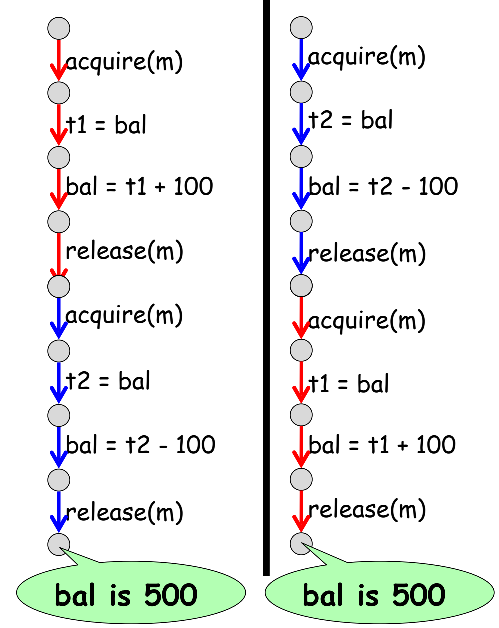
`m` is a "mutual exclusion lock"

must acquire `m` before using `bal`

Thread Interleavings

```
Thread A
  acquire(m);
   t1 = bal;
   bal = t1 + 100;
  release(m);
```

```
Thread B
  acquire(m);
   t2 = bal;
   bal = t2 - 100;
  release(m);
```



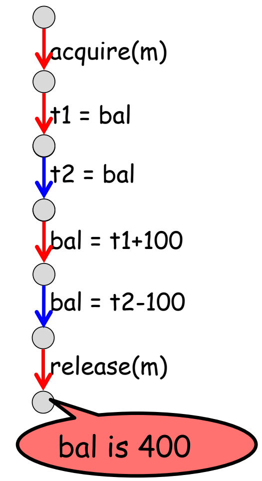
Lock downside

Unfortunately, locks are not automatic. Programmers must identify the regions of code that must be protected (called a *critical section*) and manually insert locks.

Mutex Programming Bug

```
Thread A
  acquire(m);
   t1 = bal;
   bal = t1 + 100;
  release(m);
```

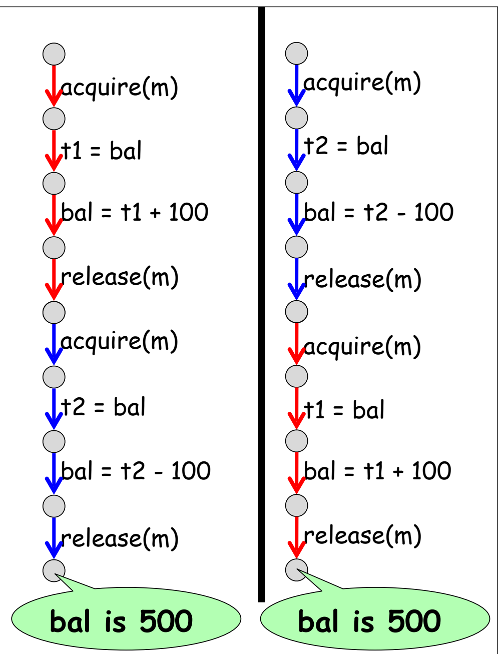
```
Thread B

   t2 = bal;
   bal = t2 – 100;
```



Common, Hard to Detect, Costly to Fix

```
Thread A
  synchronized(m){
   t1 = bal;
   bal = t1 + 100;
  }
```

```
Thread B
  synchronized(m){
   t2 = bal;
   bal = t2 – 100;
  }
```



Type Inference to Identify Races

Thread 1

```
synchronized(l) {
  x = 10;
}
synchronized(m) {
  synchronized(l) {
    x = y + 1;
  }
  y = 2;
}
```

Thread 2

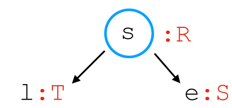
```
synchronized(m) {
  print y;
}
synchronized(m) {
  print x;
}
```

Lock Type Inference Steps

1. Get abstract syntax tree
2. Label nodes with type labels
3. Generate constraints
4. Solve constraints
5. Type check: check that use is consistent

Lock Type Inference Constraints

sync:



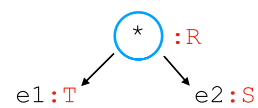

```
T = R
S = R U {l}
```

root:

if R is root

```
R = ∅
```

other:



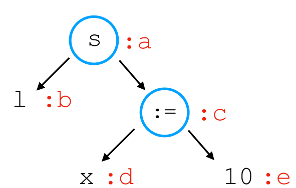

```
T = R
S = R
```

lockset:

```
i:V
lockᵢ ∈ V
```

Infer Type

```
synchronized(l) {
  x = 10;
}
```




```
a = ∅
b = a
c = a U {l}
d = c
e = c
lockₓ ∈ d
```

Infer Type

```
a = ∅
b = a
c = a U {l}
d = c
e = c
lockₓ ∈ d
```

```
a = b = ∅
c = d = e = a U {l} = {l}
lockₓ ∈ {l}
```

`x` has lock type `l`
i.e., `x` is guarded by lock `l`

Activity

```
synchronized(m) {
  synchronized(l) {
    x = y + 1;
  }
  y = 2;
}
```

What are lock types for `x` and `y`?
Are they consistent with previous example?

Consistency

```
synchronized(l) {
  x = 10;
}
synchronized(m) {
  synchronized(l) {
    x = y + 1;
  }
  y = 2;
}
```

$lock_{x1} \in \{l\}$

$lock_{x2} \in \{l,m\}$

$lock_{y1} \in \{l,m\}$

$lock_{y2} \in \{m\}$

$lock_{x1} \cap lock_{x2} = \{l\}$ $lock_{y1} \cap lock_{y2} = \{m\}$

Lock set for variable should never be empty!

Type Inference to Identify Races

Thread 1

```
synchronized(l) {
  x = 10;        {l}
}
synchronized(m) {
  synchronized(l) {
{l,m}  x = y + 1;
                {l,m}
  }
  y = 2;
     {m}
}
```

Thread 2

```
synchronized(m) {
  print y;   {m}
}
synchronized(m) {
  print x;   {m}
}
```

What's the problem here?

Call Stack

A *call stack* is a control structure that stores information about the active subroutines of a program.

Most programming language runtimes use a call stack to evaluate a program instead of evaluation-by-substitution (i.e., **λ**-calculus reductions).

GC example from HW2

```
(car (cdr (cons (cons a b) (cons c b))))
```

GC example from HW2

```
(car (cdr (cons (cons a b) (cons c b))))

                 a  b  c  d
```

main x

call stack

GC example from HW2

```
(car (cdr (cons (cons a b) (cons c b))))

                 a  b  c  d
```

car x

main x

call stack

GC example from HW2

```
(car (cdr (cons (cons a b) (cons c b))))

                 a  b  c  d
```

cdr x

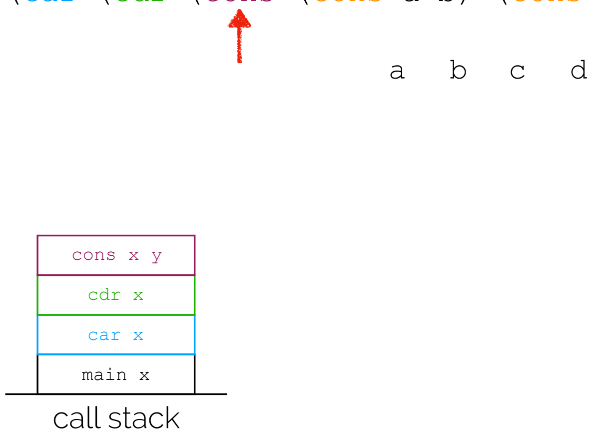
car x

main x

call stack

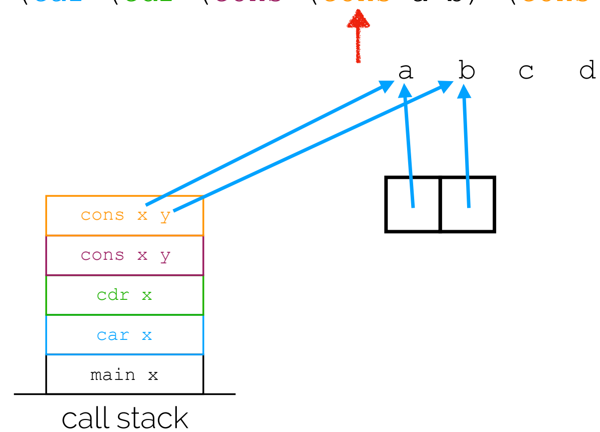
GC example from HW2

`(car (cdr (cons (cons a b) (cons c b))))`



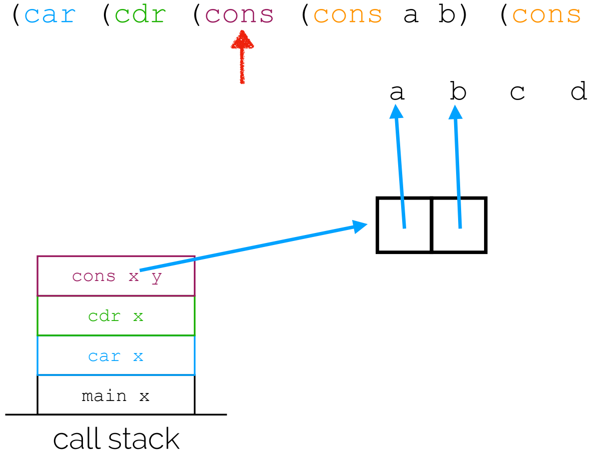
GC example from HW2

`(car (cdr (cons (cons a b) (cons c b))))`



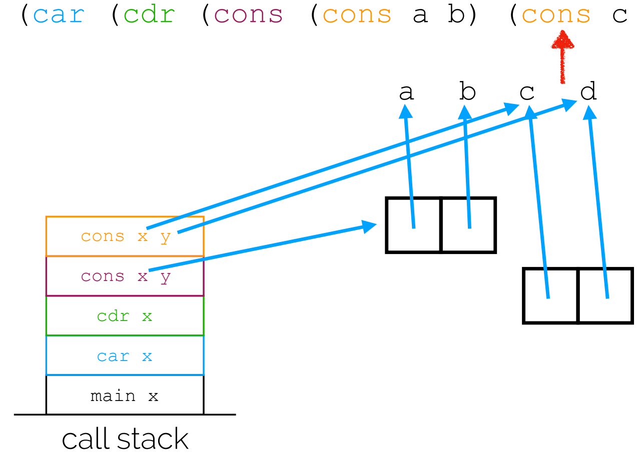
GC example from HW2

`(car (cdr (cons (cons a b) (cons c b))))`



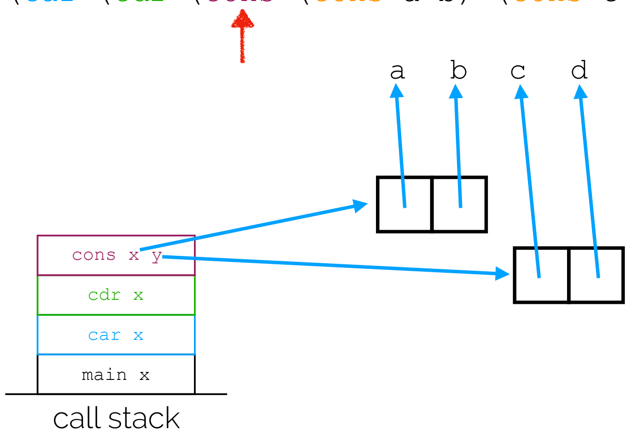
GC example from HW2

`(car (cdr (cons (cons a b) (cons c b))))`



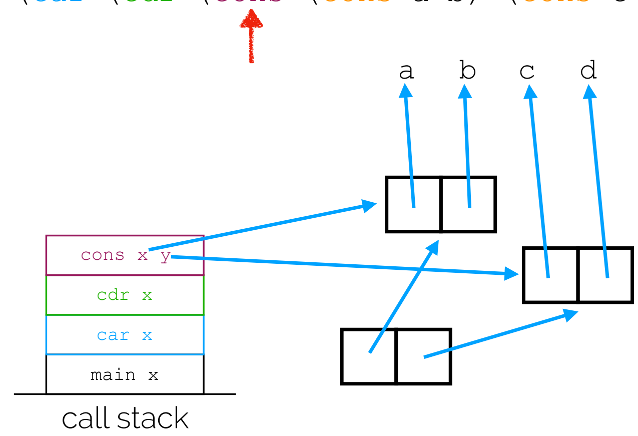
GC example from HW2

`(car (cdr (cons (cons a b) (cons c b))))`



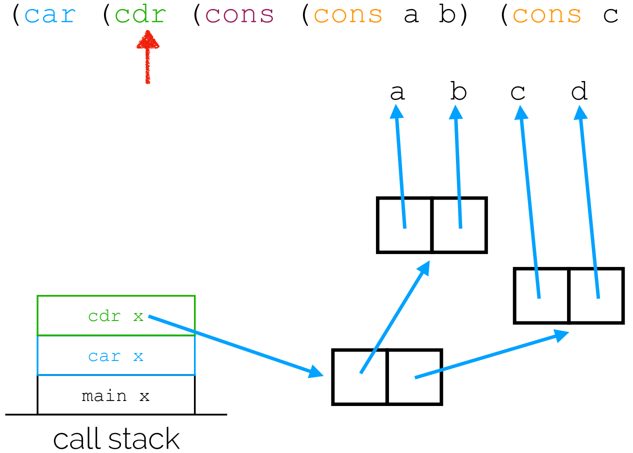
GC example from HW2

`(car (cdr (cons (cons a b) (cons c b))))`



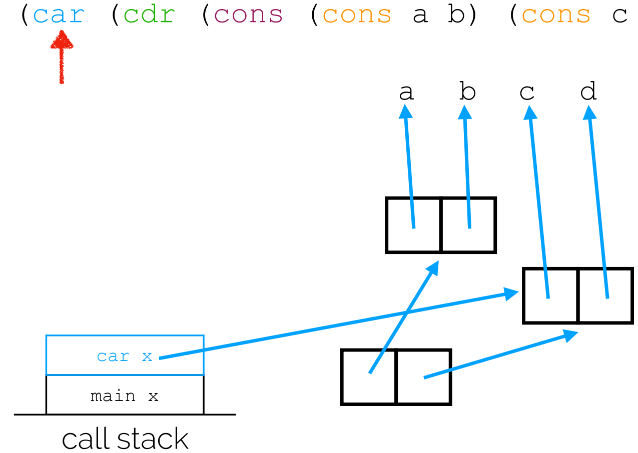
GC example from HW2

`(car (cdr (cons (cons a b) (cons c b))))`



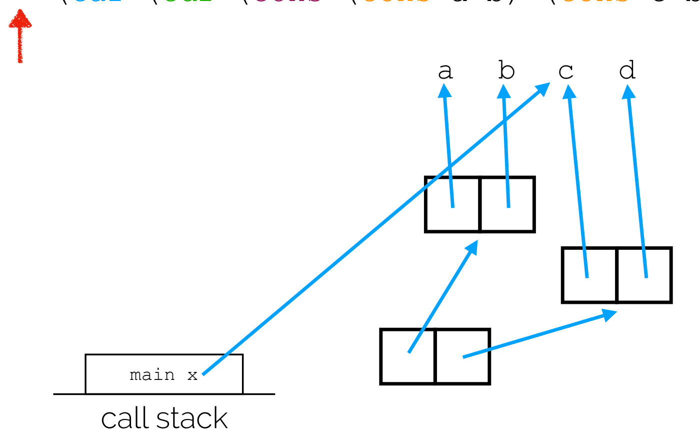
GC example from HW2

`(car (cdr (cons (cons a b) (cons c b))))`



GC example from HW2

```
(car (cdr (cons (cons a b) (cons c b))))
```



GC example from HW2

```
(car (cdr (cons (cons a b) (cons c b))))
```