

Programs as Data
Real-world abstract machines for
Java and C#/.NET.
Garbage collection techniques

Peter Sestoft
Monday 2011-10-24



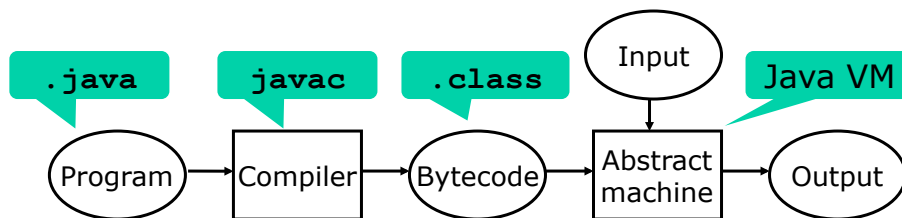
Microsoft Techtalk 9. november

- Simon Peyton-Jones
- Onsdag 9. november 14:00-16:00
- Microsoft Development Center Copenhagen
Frydenlunds allé 6, Vedbæk
- Simon er hovedmanden bag sproget Haskell
og meget andet, og en fremragende taler
- Modtager af ACM SIGPLAN's
programmeringssprogpris 2011
- Emner:
 - Typeklasser i Haskell
 - Typefunktioner i Haskell
- Tilmelding: <https://msevents.microsoft.com/CUI/EventDetail.aspx?EventID=1032496392&Culture=da-DK>



Today

- Java Virtual Machine
- .NET Common Language Infrastructure (CLI)
- Garbage collection (GC) techniques
 - Reference-counting
 - Mark-sweep
 - Two-space stop and copy
 - The garbage collectors in JVM and .NET
- List-C, a version of Micro-C with a heap and GC



Example Java program (ex6java.java)

```
class Node extends Object {
    Node next;
    Node prev;
    int item;
}
```

```
class LinkedList extends Object {
    Node first, last;

    void addLast(int item) {
        Node node = new Node();
        node.item = item;
        if (this.last == null) {
            this.first = node;
            this.last = node;
        } else {
            this.last.next = node;
            node.prev = this.last;
            this.last = node;
        }
    }

    void printForwards() { ... }
    void printBackwards() { ... }
}
```

JVM class file (LinkedList.class)

header	LinkedList extends Object
constant pool	<pre>#1 Object.<init>() #2 class Node #3 Node.<init>() #4 int Node.item #5 Node LinkedList.last #6 Node LinkedList.first #7 Node Node.next #8 Node Node.prev #9 void InOut.print(int)</pre>
fields	<pre>first (#6) last (#5)</pre>
methods	<pre><init>() void addLast(int) void printForwards() void printBackwards()</pre>
class attributes	source "ex6java.java"

Generated by
javac ex6java.java

Shown by
javap -c LinkedList

```
Stack=2, Locals=3, Args_size=2
0 new #2 <Class Node>
3 dup
4 invokespecial #3 <Method Node.<init>()>
7 astore_2
8 aload_2
9 iload_1
10 putfield #4 <Field int item>
13 ...
```

Some JVM bytecode instructions

Kind	Example instructions
push constant	iconst, ldc, aconst_null, ...
arithmetic	iadd, isub, imul, idiv, irem, ineg, iinc, fadd, ...
load local variable	iload, aload, fload, ...
store local variable	istore, astore, fstore, ...
load array element	iaload, baload, aaload, ...
stack manipulation	swap, pop, dup, dup_x1, dup_x2, ...
load field	getfield, getstatic
method call	invokestatic, invokevirtual, invokespecial
method return	return, ireturn, areturn, freturn, ...
unconditional jump	goto
conditional jump	ifeq, ifne, iflt, ifle, ...; if_icmpeq, if_icmpne, ...
object-related	new, instanceof, checkcast

Type prefixes: i=int, a=object, f=float, d=double, s=short, b=byte, ...

JVM bytecode verification

The JVM bytecode is *statically verified* before execution:

- An instruction must work on stack operands and local variables of the correct type
- A method must use no more local variables and no more local stack positions than it claims to
- For every point in the bytecode, the local stack has the same depth whenever that point is reached
- A method must throw no more exceptions than it admits to
- The execution of a method must end with a return or throw instruction, not 'fall off the end'
- Execution must not use one half of a two-word value (e.g. a long) as a one-word value (int)

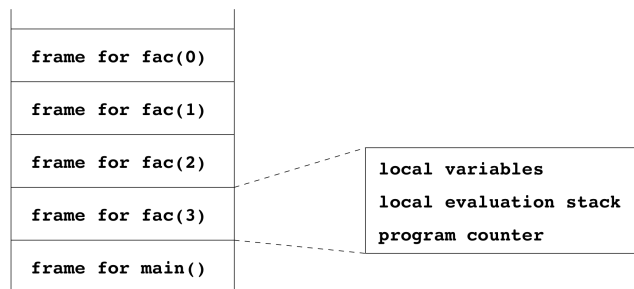
Additional JVM runtime checks

- Array-bounds checks
- Array assignment checks: Can store only subtypes of A into an A-array
- Null-reference check (a reference is null or points to an object or array, because no pointer arithmetics)
- Checked casts: Cannot make arbitrary conversions between object classes
- Memory allocation succeeds or throws exception
- No manual memory deallocation or reuse

- Bottom line: A JVM program cannot read or overwrite arbitrary memory
- Better debugging, better security
- No buffer overflow attacks, worms, etc as in C/C++

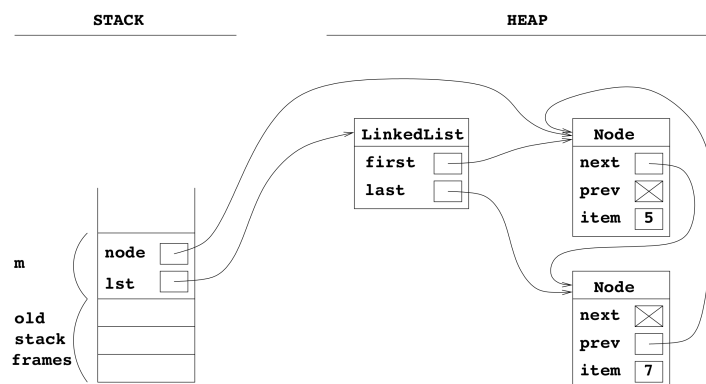
The JVM runtime stacks

- One runtime stack per thread
 - Contains activation records, one for each active function call
 - Each activation record has program counter, local variables, and local stack for intermediate results



Example JVM runtime state

```
void m() {  
    LinkedList lst = new LinkedList();  
    lst.addLast(5);  
    lst.addLast(7);  
    Node node = lst.first;  
}
```



The .NET Common Language Infrastructure (CLI, CLR)

- Same philosophy and design as JVM
- Some improvements:
 - Standardized bytecode assembly format
 - Better versioning, strongnames, ...
 - Designed as target for multiple source languages (C#, VB.NET, JScript, Eiffel, F#, Python, Ruby, ...)
 - User-defined value types (structs)
 - Tail calls to support functional languages
 - True generic types in bytecode: safer, more efficient, and more complex
- The .exe = stub + bytecode
- Standardized as Ecma-335



Some .NET CLI bytecode instructions

Kind	Example instructions
push constant	ldc.i4, ldc.r8, ldnull, ldstr, ldtoken
arithmetic	add, sub, mul, div, rem, neg; add.ovf, sub.ovf, ...
load local variable	ldloc, ldarg
store local variable	stloc, starg
load array element	ldelem.i1, ldelem.i2, ldelem.i4, ldelem.r8
stack manipulation	pop, dup
load field	ldfld, ldstfld
method call	call, calli, callvirt
method return	ret
unconditional jump	br
conditional jump	brfalse, brtrue; beq, bge, bgt, ble, blt, ...; bge.un ...
object-related	newobj, isinst, castclass

Type suffixes: i1=byte, i2=short, i4=int, i8=long, r4=float, r8=double, ...

Canonical compilation?

- Consider the Java/C#/C program ex13:

```
static void Main(string[] args) {
    int n = int.Parse(args[0]);
    int y;
    y = 1889;
    while (y < n) {
        y = y + 1;
        if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0)
            InOut.PrintI(y);
    }
    InOut.PrintC(10);
}
```

- Let us compile and disassemble it twice:
 - javac ex13.java then javap -c ex13
 - csc /o ex13.cs then ildasm /text ex13.exe



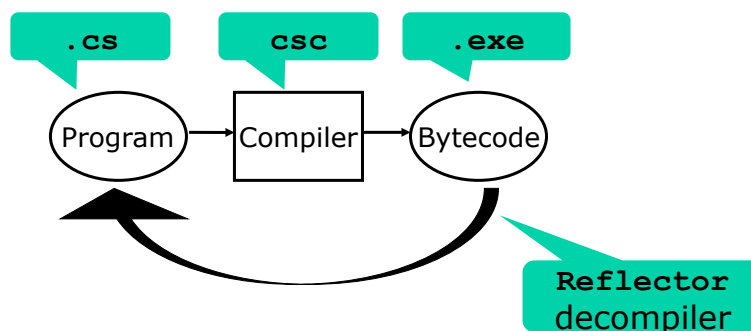
0	aload_0		IL_0000:	ldarg.0		args
1	iconst_0		IL_0001:	ldc.i4.0		
2	aaload		IL_0002:	ldelem.ref		args[0]
3	invokestatic #2 (...)		IL_0003:	call (...)		parse int
6	istore_1		IL_0008:	stloc.0		n = ...
7	sipush 1889		IL_0009:	ldc.i4 0x761		
10	istore_2		IL_000e:	stloc.1		y = 1889;
11	goto 43		IL_000f:	br.s IL_002f		while (...) {
14	iload_2		IL_0011:	ldloc.1		
15	iconst_1		IL_0012:	ldc.i4.1		
16	iadd		IL_0013:	add		
17	istore_2		IL_0014:	stloc.1		y = y + 1;
18	iload_2		IL_0015:	ldloc.1		
19	iconst_4		IL_0016:	ldc.i4.4		
20	irem		IL_0017:	rem		
21	ifne 31		IL_0018:	brtrue.s IL_0020		y % 4 == 0
24	iload_2		IL_001a:	ldloc.1		
25	bipush 100		IL_001b:	ldc.i4.s 100		
27	irem		IL_001d:	rem		
28	ifne 39		IL_001e:	brtrue.s IL_0029		y % 100 != 0
31	iload_2		IL_0020:	ldloc.1		
32	sipush 400		IL_0021:	ldc.i4 0x190		
35	irem		IL_0026:	rem		
36	ifne 43		IL_0027:	brtrue.s IL_002f		y % 400 == 0
39	iload_2		IL_0029:	ldloc.1		
40	invokestatic #3 (...)		IL_002a:	call (...)		print y
43	iload_2		IL_002f:	ldloc.1		
44	iload_1		IL_0030:	ldloc.0		
45	if_icmplt 14		IL_0031:	blt.s IL_0011		(y < n) }
48	bipush 10		IL_0033:	ldc.i4.s 10		
50	invokestatic #4 (...)		IL_0035:	call (...)		newline
53	return		IL_003a:	ret		return

Ten-minute exercise

- On a printout of the preceding slide
 - Draw arrows to indicate where jumps go
 - Draw blocks around the bytecode segments corresponding to fragments of the Java/C# program

Metadata and decompilers

- The .class and .exe files contains *metadata*: names and types of fields, methods, classes
- One can *decompile* bytecode into programs:



- Bad for protecting your secrets (intellectual property)
- *Bytecode obfuscators* make decompilation harder

.NET CLI has generic types, JVM doesn't

```
class CircularQueue<T> {  
    private readonly T[] items;  
    public CircularQueue(int capacity) {  
        this.items = new T[capacity];  
    }  
    public T Dequeue() { ... }  
    public void Enqueue(T x) { ... }  
}
```

Source;
generics

```
.class CircularQueue`1<T> ... {  
    .field private readonly !T[] items  
    ...  
    .method !T Dequeue() { ... }  
    .method void Enqueue(!T x) { ... }  
}
```

.NET CLI;
generics

```
class CircularQueue ... {  
    public java.lang.Object dequeue(); ...  
    public void enqueue(java.lang.Object); ...  
}
```

JVM; **no**
generics

17

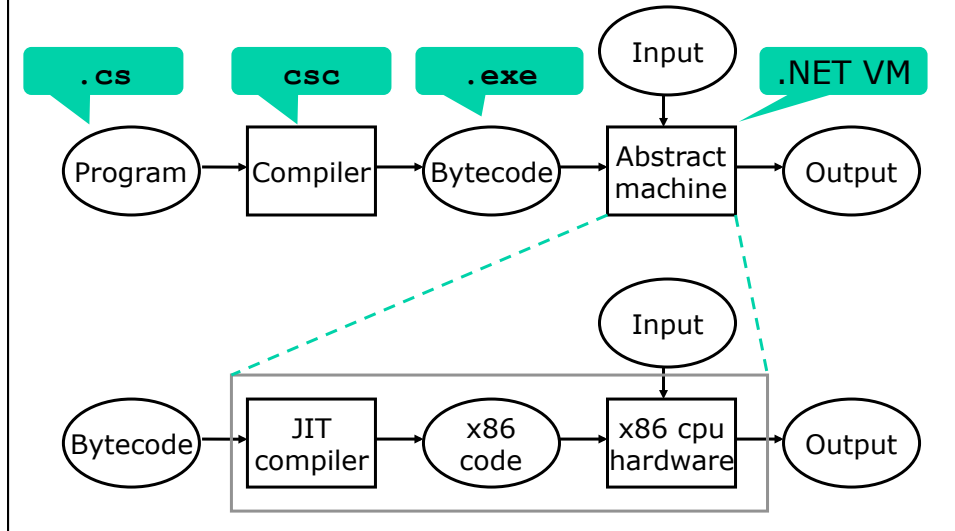
Consequences for Java

- The Java compiler replaces T
 - with `Object` in `C<T>`
 - with `Mytype` in `C<T extends Mytype>`
- So this *doesn't* work in Java, but works in C#:
 - Cast: `(T)e`
 - Instance check: `(e instanceof T)`
 - Reflection: `T.class`
 - Overload on different type instances of gen class:

```
void put(CircularQueue<Double> cq) { ... }  
void put(CircularQueue<Integer> cq) { ... }
```
 - Array creation: `arr=new T[10]`
So Java versions of `CircularQueue<T>` must use `ArrayList<T>`, not `T[]`

Just-in-time (JIT) compilation

- Bytecode is compiled to real (e.g. x86) machine code at runtime to get speed comparable to C/C++



Just-in-time compilation

- How to inspect .NET JITted code

```
csc /debug /o Square.cs
```

```
static double Sqr(double x) {
    return x * x;
}
```

C#

```
IL_0000: ldarg.0
IL_0001: ldarg.0
IL_0002: mul
IL_0003: ret
```

CLI

.NET JIT compiler

Microsoft mdbg
no longer works

Try mono -v -v

Function Square.Sqr (at 0x14a5ed0)

x86

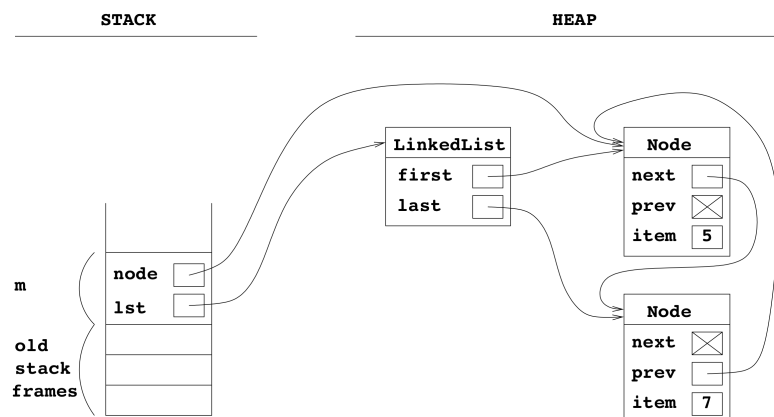
```
[0000] push    ebp
[0001] mov     ebp,esp
[0003] cmp    dword ptr ds:[00B82E28h],0
[000a] je     00000007
[000c] call   78C247E5
*[0011] fld    qword ptr [ebp+8]
[0014] fmul   st,st(0)
[0016] pop    ebp
[0017] ret    8
```

Garbage collection

- A: Reference counting
- B: Mark-sweep
- C: Two-space stop-and-copy, compacting
- Generational
- Conservative

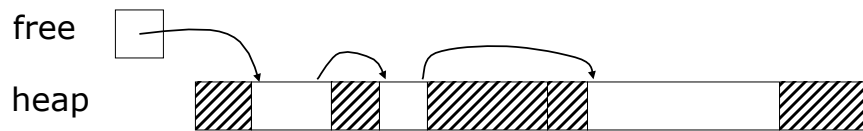
The heap as a graph

- The heap is a *graph*: node=object, edge=reference
- An object is *live* if reachable from *roots*
- Garbage collection *roots* = stack elements



The freelist

- A freelist is a linked list of free heap blocks:



- Allocation from freelist:
 - Search for a large enough free block
 - If none found, do garbage collection
 - Try the search again
 - If it fails, we are out of memory



A: Reference counting with freelist

- Each object knows the number of references to it
- Allocate objects from the freelist
- After assignment $x=o$; the runtime system
 - Increments the count of object o
 - Decrements the count of x 's old reference (if any)
 - If count becomes zero,
 - put this object on the freelist
 - recursively decrement count of all objects this one points to
- Good
 - Simple to implement
- Bad
 - Reference count updates and checks are slow
 - A cascade of decrements takes long time, gives long pause
 - Cannot deallocate cyclic structures

B: Mark-sweep with freelist

- Allocate objects from the freelist
- GC phase 1: mark phase
 - Assume all objects are white to begin with
 - Find all objects that are reachable from the stack, and color them black
- GC phase 2: sweep phase
 - Scan entire heap, put all white objects on the freelist, and color black objects white
- Good
 - Rather simple to implement
- Bad
 - Sweep must look at entire heap, also dead objects; inefficient when many small objects die young
 - Risk of *heap fragmentation*



C: Two-space stop and copy

- Divide heap into to-space and from-space
- Allocate objects in from-space
- When full, recursively move all reachable objects from from-space to (empty) to-space
- Swap (empty) from-space with to-space
- Good
 - Need only to look at live objects
 - Good reference locality and cache behavior
 - Compacting: no fragmentation
- Bad
 - Uses twice as much memory as maximal live object size
 - Need to update references when moving objects
 - Moving a large object (e.g. an array) is slow
 - Very slow (much copying) when heap is nearly full



D: Generational garbage collection

- Observation: Most objects die young
- Divide heap into *young* (nursery) and *old* generation
- Allocate in young generation
- When full, move live objects to old gen. (minor GC)
- When old gen. full, perform a (major) GC there
- Good
 - Recovers much garbage fast
- Bad
 - May suffer fragmentation of old generation (if mark-sweep)
 - Needs a write barrier test on field assignments:
After assignment $o.f=y$ where o in old and y in young, need to remember that y is live

Conservative garbage collectors

- Is 0xFFFFFFFFFA on the stack an int or a heap ref?
- If the GC doesn't know, it must be *conservative*:
Assume it could be a reference to an object
- Conservative collectors exist as C/C++ libraries
- Good
 - Can be added to C and C++ programs as a library
 - Works even with pointer arithmetics
- Bad
 - Unpredictable memory leaks
 - Cannot be compacting: updating a "reference" that is actually a customer number leads to madness

Concurrent garbage collection

- In a multi-cpu machine, let one cpu run GC
- Complicated
 - Race conditions when allocating objects
 - Race conditions when moving objects
- Typically suspends threads at GC safe points
 - May considerably reduce concurrency (because one thread may take long to reach a safe point)



GC in mainstream virtual machines

- Sun Hotspot JVM (client+server)
 - Three generations
 - When gen. 0 is full, move live objects to gen. 1
 - Gen. 1 uses two-space stop-and-copy GC; when objects get old they are moved to gen. 2
 - Gen. 2 uses mark-sweep with compaction
- IBM JVM (used in e.g. Websphere server)
 - Highly concurrent generational; see David Bacon's paper
- Microsoft .NET (desktop+server)
 - Three generation small-obj heap + large-obj heap
 - When gen. 0 is full, move to gen. 1
 - When gen. 1 is full, move to gen. 2
 - Gen. 2 uses mark-sweep with occasional compaction
- Mono .NET implementation
 - Boehm's conservative collector (still standard Oct 2011)
 - New two-generational (stop-and-copy plus M-S or S-A-C)



Other GC-related topics

- *Large object space*: Large arrays and other long-lived objects may be stored separately
- *Weak reference*: A reference that cannot itself keep an object live
- *Finalize*: Code that will be executed when an object dies and gets collected (e.g. close file)
- *Resurrection*: A finalizer may make a dead object live again (yrk!)
- *Pinning*: When Java/C# exports a reference to C/C++ code, the object must be pinned; if GC moves it, the reference will be wrong

GC stress (StringConcatSpeed.java)

- What do these loops do? Which is better?

```
StringBuilder buf
= new StringBuilder();
for (int i=0; i<n; i++)
    buf.append(ss[i]);
res = buf.toString();
```

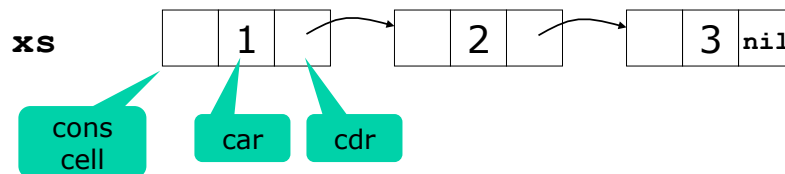
```
String res = "";
for (int i=0; i<n; i++)
    res += ss[i];
```


New: List-C and the list machine

- list-c = micro-C with Lisp/Scheme data

```
void main(int n) {
  dynamic xs;
  xs = nil;
  while (n>0) {
    xs = cons(n,xs);
    n = n - 1;
  }
  printlist(xs);
}
```

```
void printlist(dynamic xs) {
  while (xs) {
    print car(xs);
    xs = cdr(xs);
  }
}
```



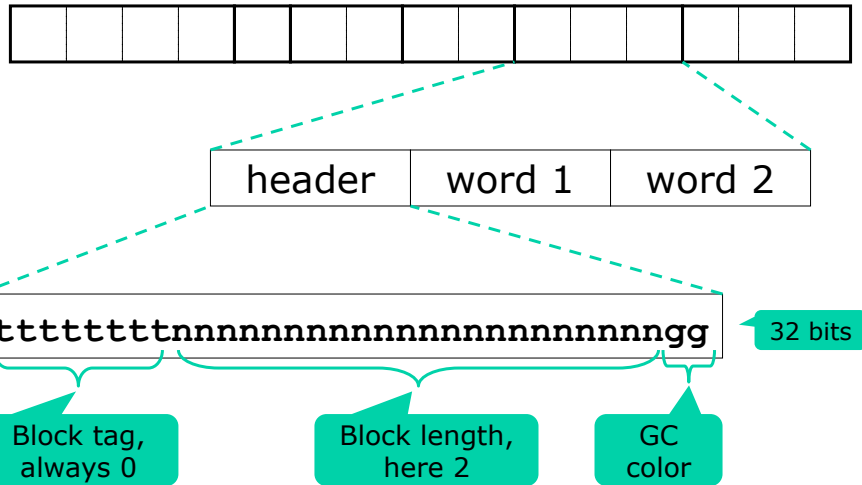
List machine instructions

- List machine = micro-C abstract machine plus six extra instructions:
 - NIL: Put nil reference on stack
 - CONS: Allocate two-word block on heap, put reference to it on stack
 - CAR, CDR: Access word 1 or 2 of block
 - SETCAR, SETCDR: Set word 1 or 2 of block

Instr	St before	St after	Effect
26 NIL	s	$\Rightarrow s, nil$	Load <i>nil</i> reference
27 CONS	s, v_1, v_2	$\Rightarrow s, p$	Create cons cell $p \mapsto (v_1, v_2)$ in heap
28 CAR	s, p	$\Rightarrow s, v_1$	Component 1 of $p \mapsto (v_1, v_2)$ in heap
29 CDR	s, p	$\Rightarrow s, v_2$	Component 2 of $p \mapsto (v_1, v_2)$ in heap
30 SETCAR	s, p, v	$\Rightarrow s$	Set component 1 of $p \mapsto _$ in heap
31 SETCDR	s, p, v	$\Rightarrow s$	Set component 2 of $p \mapsto _$ in heap

The structure of the list machine heap

- The heap consists of 32-bit (4-byte) *words*
- The heap is covered by *blocks*



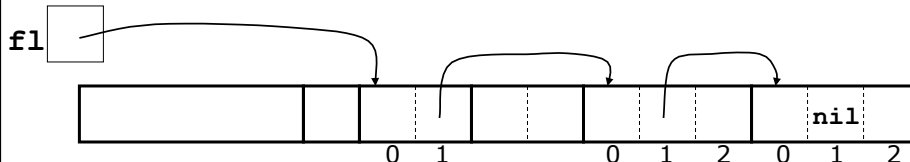
Garbage collection bits gg

Bits	Color	Meaning
00	white	After mark phase: Not reachable from stack; may be collected
01	grey	During mark phase: Reachable, referred-to blocks not yet marked
10	black	After mark phase: Reachable from stack; cannot be collected
11	blue	On freelist, or is orphan block

- The *mark phase* paints all reachable blocks black
- The *sweep phase* paints black blocks white; paints white blocks blue and puts them on freelist

The freelist; orphans

- A block on the freelist is blue (gg=11)
- Word 1 contains a reference to the next freelist element, or nil:



- A block of length zero is an *orphan*
- It consists of a header only
- Cannot be on freelist: no room for next ref.
- (Created by allocating almost all of a block)



Distinguishing integers and references

- For *exact* garbage collection we need to distinguish integers from references
- Old trick:
 - Make all heap blocks begin on address that is a multiple of 4; in binary it has form xxxxxx00
 - Represent integer n as $2n+1$, so the integer's representation has form xxxxxx1
- Test for `IsInt(v)`: $(v) \& 1 == 1$
- Tagging an int: $((v) \ll 1) | 1$
- Untagging an int: $(v) \gg 1$



An example list-C program, ex30.lc

- Each iteration allocates a cons cell that dies
- Without a garbage collector the program soon runs out of memory

```
void main(int n) {
    dynamic xs;
    while (n>0) {
        xs = cons(n, 22);
        print car(xs);
        n = n - 1;
    }
}
```

Assignment causes previous xs value to die

Allocate cons cell in heap

- Your task in BOSC: Implement garbage collectors: mark-sweep, and stop-and-copy

Reading and homework

- This week's lecture:
 - PLCSO chapters 9 and 10
 - Sun Microsystems: Memory Management in the Java Hotspot Virtual Machine
 - David Bacon, IBM: Realtime garbage collection
 - Exercises 9.1 and 9.2
- Next week's lecture:
 - PLCSO chapter 11