COMP 520 - Compilers

Lecture 15 (Tue Apr 12, 2022)

miniJava code generation and runtime organization

- Reading
  - skim PLPJ Chapter 8 on interpretation
  - study example from class today
  - study mJAM miniJava Abstract Machine
- PA4 project materials online
  - PA4 assignment
  - mJAM virtual machine (instead of TAM)
  - PA4Test.java

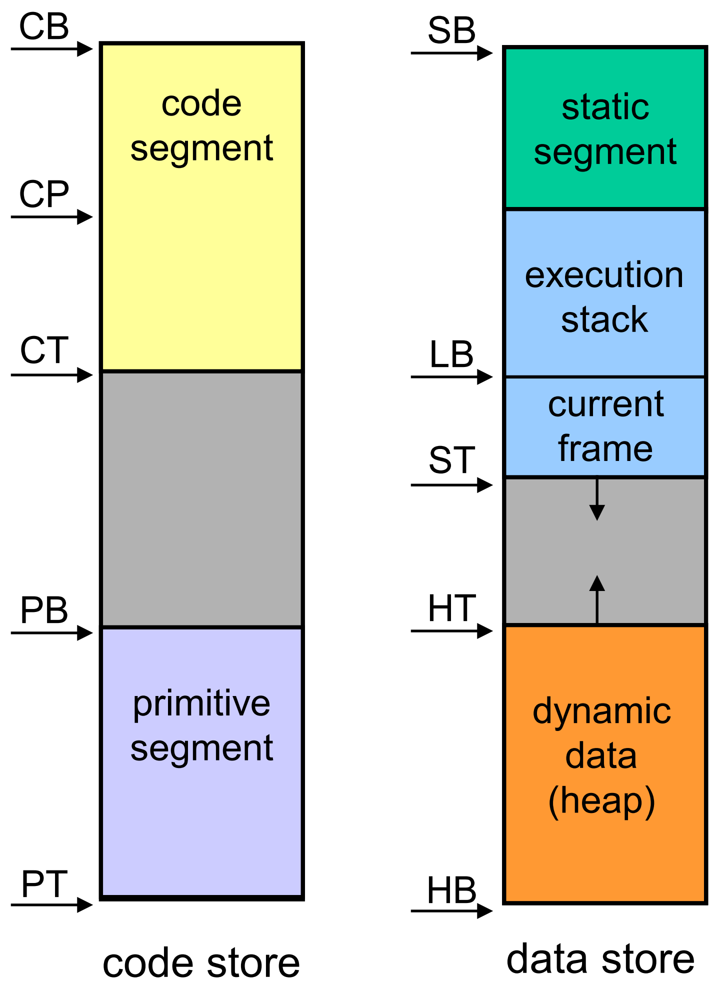
On to PA4 Code Generation

- Recall Triangle Abstract Machine (TAM)
  - TAM interprets code generated by the Triangle compiler
  - Triangle and miniJava are quite different
  - we will use mJAM, a modified version of TAM, as our target machine

- What are the differences?
  - top-level: nested procedures vs. objects

mJAM memory organization

- Two separate memories
  - Code store
    - compiler-generated program is loaded into code segment
    - predefined runtime functions are located in the primitive segment
    - mJAM can not write into code store
  - Data store
    - static constants and variables are loaded into static segment
    - method invocation creates a frame
    - expression evaluation occurs at stack top
      - expands downwards
    - object instances are dynamically allocated on the heap
      - expands upwards
      - (no garbage collection)
- ABI defines fixed addresses and usage conventions
  - various locations in memories are accessed relative to machine registers (CB, SB, LB, ST, etc.)



miniJava: simple classes, no inheritance

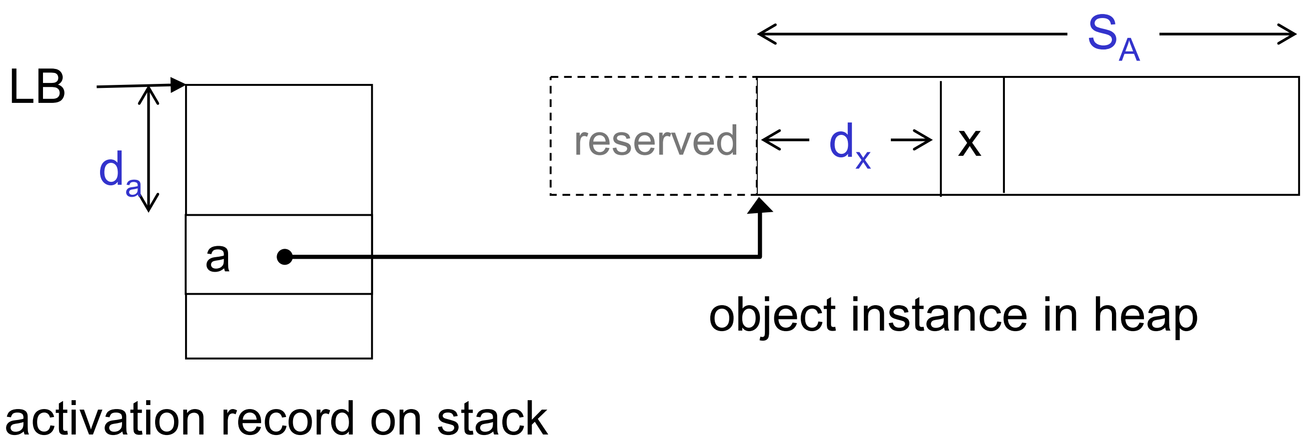
- Classes

```
class A { int x; void p(){x = 3;} }
```

  - runtime entity descriptions in AST
    - class A : S_A = size of class A (# fields) = 1
    - field x: d_x = displacement of field x = 0
    - method p: d_p = displacement of code for p = ?

- Objects
  - objects are created on the heap: `A a = new A();`
  - let d_a be displacement of local var "a" in activation record (= frame)

mJAM runtime layout



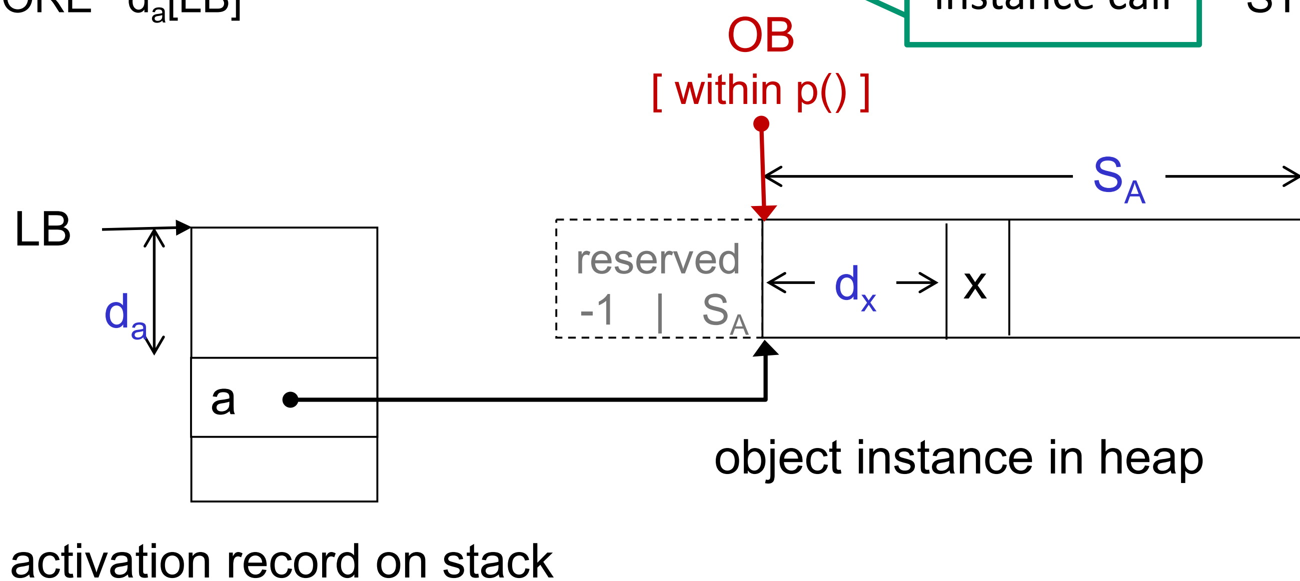
mJAM: runtime support for simple classes

- mJAM code sequences

instance address

instance call

`A a = new A();` (object creation)	`a.x;` (qualified reference)	`a.p();` (method invocation)	`x = x + 3;` (field upd within p())
LOADL -1	LOAD d_a[LB]	LOAD d_a[LB]	LOAD d_x[OB]
LOADL S_A	LOADL d_x	CALLI d_p[CB]	LOADL 3
CALL newobj	CALL fieldref		CALL ADD
STORE d_a[LB]			STORE d_x[OB]



Linkage

- In a method call, the first three words of the new frame are reserved for the linkage
  - OB: the object base, or -1 for static method, of caller (i.e. caller’s OB)
  - DL: the start of caller’s frame on the stack (i.e. caller’s LB)
  - RA: the code address to resume in caller on return

  Thus the first available location in the frame of a method is 3[LB]

- On return (#res) #args
  - the frame plus #args are popped off stack
  - #res values (0 or 1) are pushed on the stack
  - execution resumes in caller

Simple miniJava program

```
class Counter {

    public void increase(int k) {
        count = count + k;
    }

    public static void main(String [] args){
        Counter counter = new Counter();
        counter.increase(3);
        System.out.println(counter.count);
    }

    public int count;

}
```

Code generation for “Counter” example (1)

- Where do we start?
  - identify unique mainclass
    - there’s only one class and it contains a

      `public static void main(String [] args){ … }`

- Emit code to call **main** and halt on return
  - code starts at location 0 in code store
  1. create empty **args** array on heap
  2. call **main** (address L11 must be patched)
  3. on return halt with code 0

instruction addresses

mJAM instructions

```
0   LOADL   0
1   CALL    newarr
2   CALL    (L11)
3   HALT    (0)
```

Code generation for “Counter” example (2)

- Visit each class in turn, generating code for all methods
  - visit class **Counter**

1. Visit method **increase**

```
public void increase(int k) {
        count = count + k;
    }
```

```
4 L10:  LOAD   0[OB]
5       LOAD   -1[LB]
6       CALL   add
7       STORE  0[OB]
8       RETURN (0) 1
```

results (0 or 1)

method arguments

Code generation for “Counter” example (3)

- Visit method **`main (String [] args) {`**

```
Counter counter = new Counter();
counter.increase(3);
System.out.println(counter.count);
}
```

```
 9  L11: LOADL   -1
10       LOADL    1
11       CALL     newobj
12       LOADL    3
13       LOAD     3[LB]
14       CALLI     (L10)
15       LOAD     3[LB]
16       LOADL    0
17       CALL     fieldref
18       CALL     putintnl
19       RETURN (0)   1
```

address of counter instance

must be patched to address of `increase` method in code store

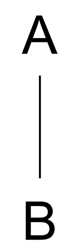
get value of `count` from our counter instance

Classes with single inheritance (Java)

- Class hierarchy

```
class A {int x; void p(){ … } }
class B extends A {int y; void p(){ … } void q(){ … } }
```

  - inheritance hierarchy
    - "class B extends class A", or "B is a subtype of A"



  - fields
    - fields of B extend the fields of A
    - runtime layout of fields in A is a prefix of the runtime layout of fields in B

  - methods
    - methods of B extend the methods of A
    - methods of B can redefine (override) methods of A

Static and dynamic type with single inheritance

- Object type
  - static type (declared type)
    - used by compiler for type checking
      - determines accessible fields and available methods on objects
      - type rules for assignments
        - » assignment: (type of RHS) must be a subtype (≤) of (type of LHS)
        - » method call: type of arg *i* must be a subtype of type of parameter *i*
  - dynamic type (run-time type)
    - generally only known at runtime
      - *part of the representation of an object*
        - » initialized at time of creation from object constructor
      - dynamic type is always a subtype of the static type (guaranteed by type system)
      - dynamic type determines which method is invoked (runtime lookup)
  - examples

```
A a = new A();
B b = new B();
A c = b;
B d = a;
a.p();
b.q();
c.p();
```

```
class A {int x; void p(){ … } }

class B extends A {
    int y;
    void p(){ … }
    void q(){ … }
}
```

```
A
|
B
```

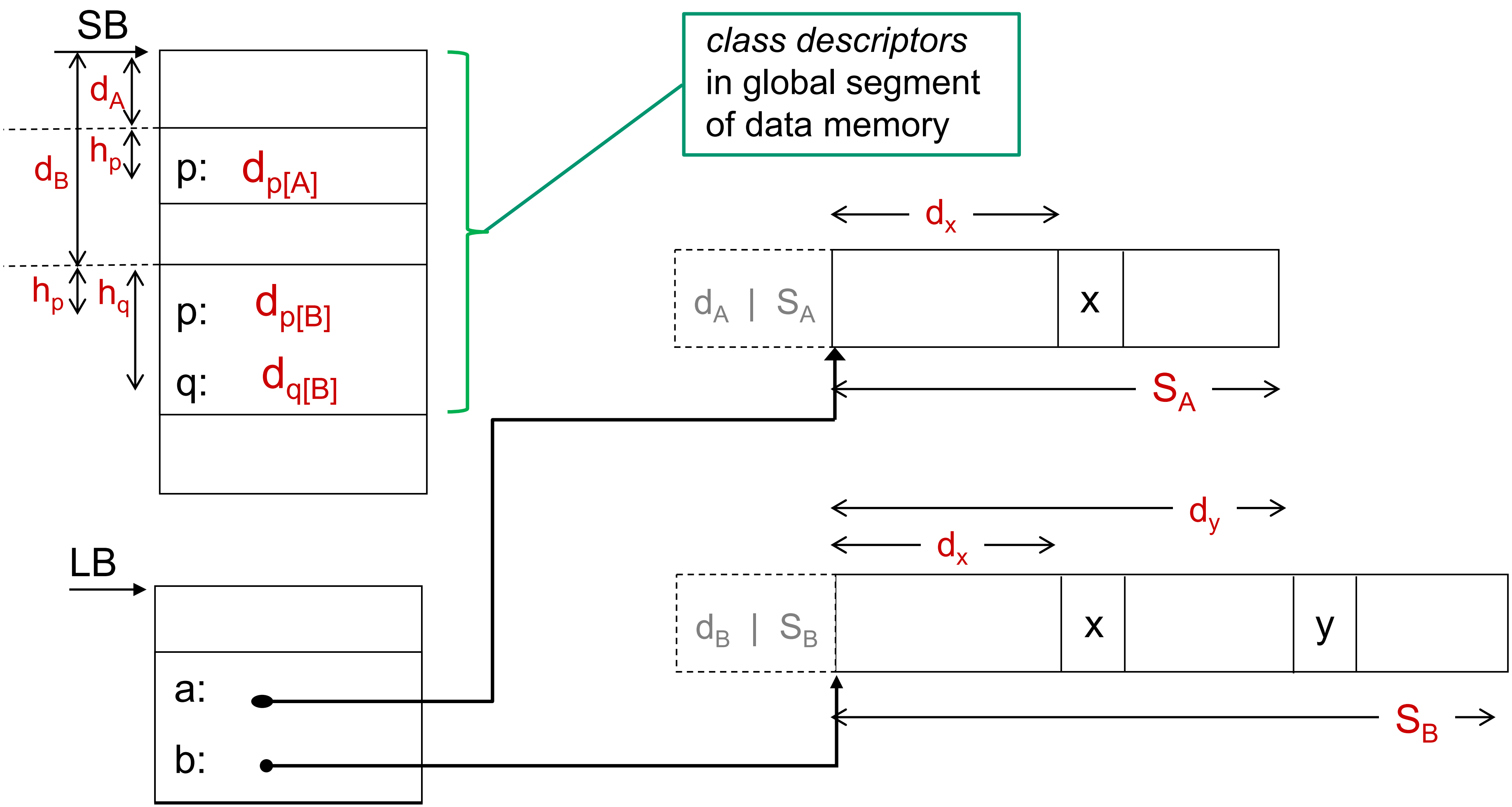
mJAM representation of single inheritance

```
class A {int x; void p(){ … } }
class B extends A
        {int y; void p(){ … } void q(){ … } }
```

- runtime entity descriptions in AST
  - class A : S_A = size of class A
  - class A: d_A = displacement of class descriptor for A
  - class B: S_B = size of class B (including size of class A)
  - class B: d_B = displacement of class descriptor for B
  - field x d_x = displacement of field x in A and B
  - field y d_y = displacement of field y in B
  - method p: h_p = index of method p in A and B
  - method q: h_q = index of method q in B
  - method p in A: $d_{p[A]}$ = displacement of code for p() in A
  - method p in B: $d_{p[B]}$ = displacement of code for p() in B
  - method q in B: $d_{q[B]}$ = displacement of code for q() in B

Classes with single inheritance

- mJAM runtime layout



Classes with single inheritance

- mJAM code sequences (only changed sequences are shown)

`A a = new A();`

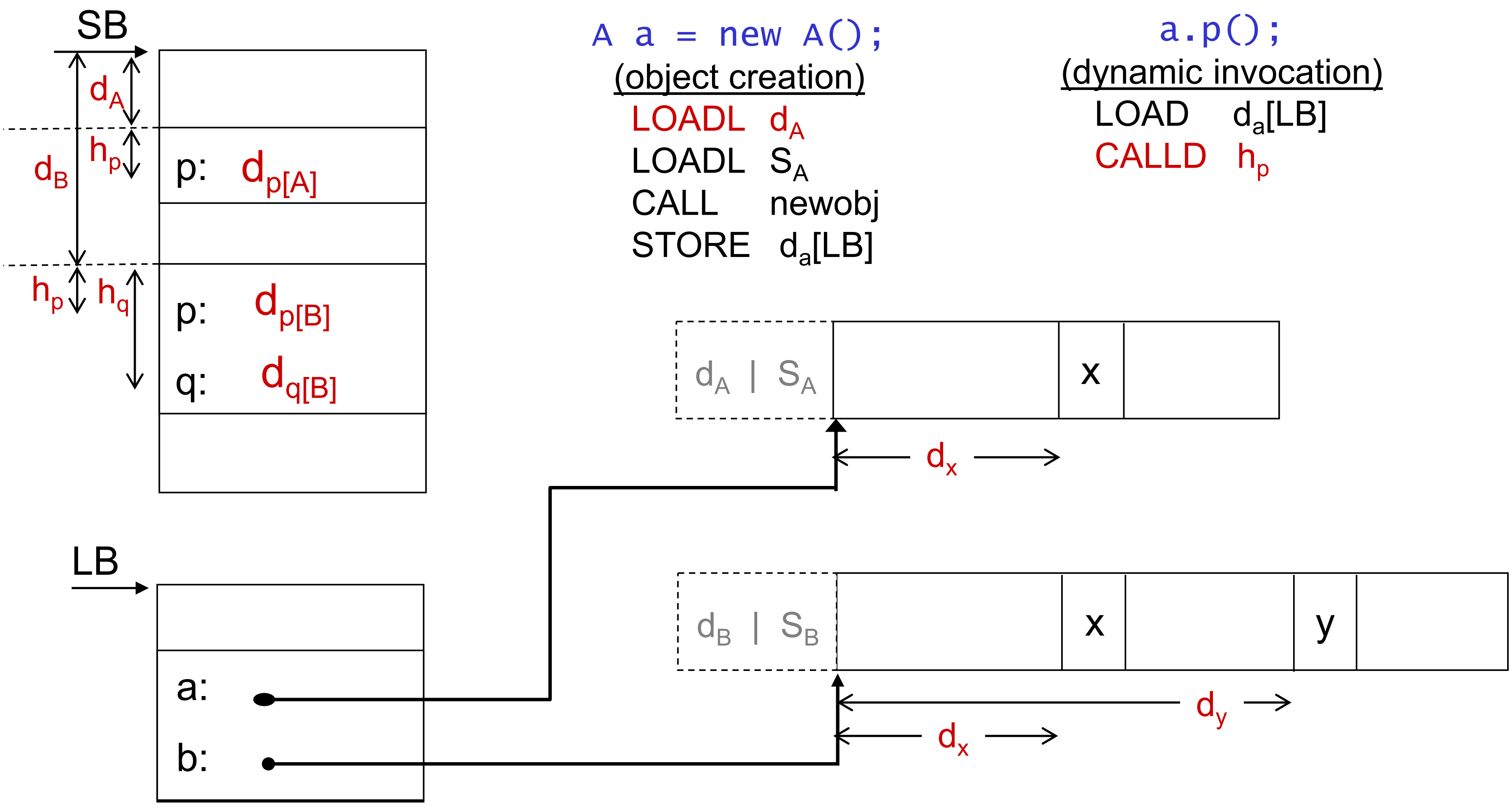
(object creation)

```
LOADL   d_A
LOADL   S_A
CALL    newobj
STORE   d_a[LB]
```

`a.p();`

(dynamic invocation)

```
LOAD    d_a[LB]
CALLD   h_p
```



Related issues

- single inheritance
  - type operations
    - instanceof
    - casting
  - `super()` superclass constructor invocation
- multiple inheritance
  - we lose the prefix property of runtime layout!
- optimization
  - dynamic method dispatch has high cost
  - converting dynamic to static calls
- dynamically loaded classes
  - Java loads classes on demand, hence cannot use simple representations such as those used by mJAM

The PA4 checkpoint

- your pa4 directory should have
  - miniJava package
    - Compiler.java
    - SyntacticAnalyzer
    - AbstractSyntaxTrees
    - ContextualAnalyzer
    - CodeGenerator (new subpackage)
  - mJAM package (supplied on our web page)
    - Interpreter.java
    - Disassembler.java
    - Instruction.java
    - Machine.java
    - ObjectFile.java
- mJAM is needed to check the generated code gives the right result
  - pa4 testing will not copy your mJAM, it uses mJAM as distributed
- pa4 readiness check will be available: `/check/pa4.pl`

Compiling and running miniJava programs (Unix)

- Compiling `test.java`
  - `java miniJava/Compiler test.java`
    - use mJAM.ObjectFile to write `test.mJAM` (note spelling!), be sure that it is written in the same directory as `test.java`
    - do not run the generated program as part of compilation!

- Disassembling `test.mJAM`
  - `java mJAM/Disassembler test.mJAM`
    - should write `test.asm` in same directory as `test.mJAM`

- Running `test.mJAM`
  - `java mJAM/Interpreter test.mJAM`
    - `System.out.println` results from test.java will appear on stdout prefixed by ">>> "

- Debugging `test.mJAM`
  - `java mJAM/Interpreter test.mJAM test.asm`
    - Show machine data store and state, show code, set/remove breakpoints, single instruction execution
    - Type "?" for help

Check results

- To compare miniJava and java semantics of program `foo.java`

1. Run as miniJava program

```
java miniJava/Compiler foo.java
java mJAM/Interpreter foo.mJAM
```

2. Run as java program

```
javac foo.java
java  foo.class
```

- Note that mJAM println prefixes output with “`>>> `“